Tektronix®
COMMITTED TO EXCELLENCE

4050 SERIES
GRAPHIC SYSTEM

REFERENCE MANUAL

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

MANUAL PART NO.
070-2056-01

First Printing JAN 1976
This Printing JUN 1979

Copyright © 1976, 1979 by Tektronix, Inc., Beaverton, Oregon. Printed in the United States of America. All rights reserved. Contents of this publication may not be reproduced in any form without permission of Tektronix, Inc.

This instrument, in whole or in part, may be protected by one or more U.S. or foreign patents or patent applications. Information provided on request by Tektronix, Inc., P.O. Box 500, Beaverton, Oregon 97077.

TEKTRONIX is a registered trademark of Tektronix, Inc.

PRODUCT 4051, 4052, 4054 Graphic Computing Systems

This manual supports the following versions of this product:

- 4051 Serial Nos. B010101 and up
- 4052 Serial Nos. B010101 and up
- 4054 Serial Nos. B010101 and up

MANUAL REVISION STATUS

REV.	DATE	DESCRIPTION
@	1/76	Original Issue
@	3/79	New pages
A	3/79	Revised
B	3/79	Revised
C	3/79	Revised
D	3/79	Revised
A,B,C,D,E	7/79	Revised

WE KNOW YOU'RE ANXIOUS TO LEARN ALL ABOUT THE GRAPHIC SYSTEM, BUT...

you'll miss valuable information if you don't start at the beginning! No matter what your objectives are, you should begin by reading the introduction in the Graphic System Operator's Manual. It presents an overview of the complete Graphic System documentation package, and it will help you select the study material you need to use the Graphic System effectively.

CONTENTS

Section 7 (cont)

Section 8 **MATH OPERATIONS**

PREFACE

About the Language

The Graphic System BASIC language is a version of time-shared BASIC with extensions in the areas of graphics, file system access, unified handling of input/output operations, matrices, character string manipulation, high level language interrupt handling, and operating system facilities.

Although these extensions give the language a power far beyond most BASIC languages, most of the extensions are exercised through optional entries in each statement. This allows the Graphic System BASIC language to be compatible with most other BASIC languages by simply leaving out these optional entries.

While most keywords of the Graphic System BASIC language are available for all members of the 4050 Series family, a few keywords are available only on the 4054 Graphic System, are not available on the 4051 Graphic System, or require a special ROM pack. If any restrictions apply to a particular keyword, they are defined in the description of that keyword.

The Graphic System BASIC language differs from other BASIC languages in that most keywords and their parameters can be evaluated independently of program control. This allows the keyboard operator to draw a vector on the display with the DRAW statement, for example, without placing the system under program control.

Almost all of the statements in the language are executed immediately if the statement is entered without a line number and the RETURN key is pressed. If a statement is preceded by a line number, however, the statement is stored in memory as a program instruction to be executed at a later time. These statements are executed sequentially when the system is placed under program control.

About the Manual

This manual documents every programmable feature of the 4050 Series Graphic System in detail. The purpose of the manual is not to teach you how to program in BASIC; this manual does, however, define the characteristics of the language in such a way as to provide a sound base for programming. The manual's purpose is to serve as an indepth reference guide for the Graphic System BASIC language.

The manual is divided into twelve sections with four appendices. Each section contains BASIC statements and functions grouped together according to their primary purpose. For example, the graphics section contains all of the statements which pertain to graphics. Statements within a section are arranged in alphabetical order for quick reference use, except for Section 1. The topics in Section 1 are arranged in a logical sequence, beginning with basic definitions and ending with the more complicated topics like dimensioning variables.

Some statements are repeated in several sections throughout the manual because they are used in different applications. For example, the INPUT statement can be used in graphic operations, magnetic tape operations, and character string operations. The explanation of a statement which is repeated is slanted toward the use of that statement with the other statements in the section.

Occasionally in the manual you will find the term "Graphic System" abbreviated to simply GS.

The following is a summary of the contents of each section and appendix:

Section 1—Language Elements contains an explanation of the fundamental elements used to construct BASIC statements.

Section 2—Environmental Control explains how to set the Graphic System's internal environmental parameters.

Section 3—System Control contains statements which cause system control functions to be executed.

Section 4—Memory Management explains how to keep track of the memory space available for storing BASIC programs and data.

Section 5—Controlling Program Flow contains the statements which are used to control the flow of a BASIC program as the program executes. The fundamentals of programming like branching, looping, and executing subroutines are explained here.

Section 6—Handling Interrupts explains how to use the Graphic System's unique high-level language interrupt facility to serially poll peripheral devices on the General Purpose Interface Bus and execute peripheral service routines.

Section 7—Input/Output Operations gives an overview of the system architecture as it pertains to input and output operations. Explains the I/O addressing facility. Explains how to transfer information to and from the GS display, the GS magnetic tape unit, the GS keyboard, and external peripheral devices on the General Purpose Interface Bus.

Section 8—Math Operations contains an explanation of all the math functions available in the language.

Section 9—Graphics gives a detailed explanation of the statements used to draw graphs on the GS display screen.

Section 10—Character Strings explains how to input, manipulate, and output character strings.

Section 11—Program Editing, Debugging, and Documentation explains how to edit, debug, and document a BASIC program.

Section 12—Language Syntax explains the rules which must be followed when BASIC statements are entered into memory.

Appendix A—Error Messages. 96 error messages are listed which attempt to pinpoint the source of an error. The message numbers match the message numbers printed on the GS display when an error occurs.

Appendix B—Tables. A list of tables used throughout the manual are provided here for quick reference use.

Appendix C—Interfacing Information contains an explanation of the hardware features of the Graphic System and the General Purpose Interface Bus.

Appendix D—Glossary defines terms which might be unfamiliar to a beginning programmer.

Index—When all else fails and you can't find it, look here.

LANGUAGE ELEMENTS

Section 1

LANGUAGE ELEMENTS

INTRODUCTION TO LANGUAGE ELEMENTS

This section defines the fundamental elements and concepts used in the Graphic System BASIC language. The terms and concepts discussed in this section are used extensively throughout the rest of this manual.

REAL NUMBERS AND CHARACTER STRINGS

Real Numbers

The Graphic System BASIC interpreter treats every number as a real decimal number; that is, a number which can be negative or positive and may or may not have a fractional part. The numbers 5, 9.86, −Ø.Ø43, and 65535 are examples of real numbers.

Integers

Integers are a group of numbers within the real number category which do not have a fractional part. The numbers 1, −2, 3, and 4 are examples of integers.

Standard Notation

Real numbers written in standard notation are written with all digits displayed. For example, the number 3280000.00 is a real number written in standard notation. Imbedded spaces and commas are not allowed in the standard notation format.

Scientific Notation (E Format)

When a real number gets too big or too small to manage conveniently with standard notation, the BASIC interpreter converts the number to scientific notation. Numbers written in scientific notation have a fractional part called the mantissa and a power of ten part called the exponent. For example, the number 3.28E+6 is a number written in scientific notation; 3.28 is the mantissa and E+6 is the exponent. The number 3.28E+6 is the same number as 3.28×10^6 which is the same number as 3280000.00.

Numeric Range for the System

The numeric range for the system extends from −8.988465674E+ 307 to + 8.988465674E+ 307. Numbers within the range ±1.0E−64 are treated as though they are equal to absolute zero unless an environmental parameter is changed. (Refer to the FUZZ statement in the Environmental Control section for details.)

Numeric Accuracy

All math calculations are computed to 14 digits of accuracy. Numbers expressed in standard notation are printed with 12 digits of accuracy. Leading and trailing zeros are suppressed unless the PRINT USING form of the PRINT statement is used. (Refer to the IMAGE statement in the Input/Output Operations section for details.) Numbers expressed in scientific notation are printed to 9 digits of accuracy in the decimal part of the mantissa. Up to 11 digits of accuracy can be displayed in the mantissa if the PRINT USING form of the PRINT statement is used.

Character Strings

Character strings are any sequence of letters, numbers, and symbols enclosed in quotation marks. Character strings are some times called string constants, literal strings, literals, or just plain "strings." Normally, a character string represents a message to be printed on the GS display or a piece of written text. Digits entered as part of a character string cannot be used in math computations; they are treated just like any other symbol. The length of a character string is limited only by the size of the random access memory.

NUMERIC CONSTANTS AND STRING CONSTANTS

Numeric Constants

The term numeric constant refers to any real number entered into the system as numeric data. Only numeric data can be used in math operations. Numeric constants can be expressed in either standard notation or scientific notation and must be in the range ±8.988465674E+ 307. The plus (+) or minus (—) sign associated with the number is treated as part of the number.

String Constants

The term string constant refers to any character string of fixed length. Every string constant must be enclosed in quotation marks. The quotation marks are delimiters (separaters) and are not considered part of the string. For example, "Isn't this fun?" is a string constant of fifteen characters. The two spaces and two punctuation marks are counted as characters. The quotation marks, however, are not considered part of the string.

If quotation marks are to be part of a string constant, they are entered as double quotes inside the quotation marks used as delimiters. For example, when "The flagpole sitter suddenly screamed ""HELP!""" is printed on the GS display, the outside quotation marks are used as delimiters and are not printed; the double quotation marks around the word HELP! are printed as single quotation marks. The result is:

The flagpole sitter suddenly screamed "HELP!"

String constants can be entered into memory with the LET statement, the INPUT statement, or the READ statement. (Refer to these statements in the Character Strings section for details.)

NUMERIC VARIABLES, STRING VARIABLES, AND ARRAY VARIABLES

Numeric Variables

Numeric variables are symbols which represent numeric constants. For example, if the numeric constant 5 is assigned to the numeric variable X, and the BASIC interpreter is called upon to evaluate a statement containing the variable X, the BASIC interpreter replaces X with its assigned value (5) before the statement is evaluated. Specially, if X=5 and the BASIC interpreter evaluates the equation Y=X↑2, then the variable X is replaced with its assigned value 5; the result (25) is assigned to the numeric variable Y. If X does not have an assigned value when the equation Y=X↑2 is evaluated, an undefined variable error occurs.

There are 286 possible symbols which can be used to represent numeric constants. All twenty-six upper case letters (A-Z) are valid symbols. Also, an upper case letter followed by a digit from Ø-9 is valid. For example, A, AØ, A1, A2, A3, A4, A5, A6, A7, A8, and A9 are all valid symbols. Eleven combinations for each letter of the alphabet are possible, as shown above with the letter A, for a combined total of 286. If a lower case letter is entered as a numeric variable, the BASIC interpreter automatically converts the letter to upper case.

Numeric constants are assigned to numeric variables with the LET statement, the INPUT statement, and the READ statement. The LET statement is discussed later in the section. Refer to the Input/Output Operations section for details on the INPUT statement and the READ statement.

It is appropriate to mention at this point that numeric functions and numeric expressions can also be assigned to numeric variables, as long as the function or expression can be reduced to a numeric constant. In addition, a numeric variable can assume a succession of values over a period of time, but can represent only one value at any given time.

String Variables

String variables are symbols which represent string constants. For example, if the string constant "Isn't this fun?", is assigned to the string variable A$ and the BASIC interpreter is called upon to PRINT A$, then the BASIC interpreter prints the string constant represented by A$; in this case "Isn't this fun?" Notice here again, the quotation marks around the string constant serve only as delimiters and are not considered part of the string.

There are twenty-six symbols that can be used for string variables — upper case letters from A-Z, followed by a dollar sign. For example, A$, B$, and Z$ are valid symbols for string variables. If a lower case letter and a dollar sign are entered as a string variable symbol, the BASIC interpreter automatically converts the letter to upper case. A string variable can represent a succession of string constants over a period of time, but can only represent one string constant at any given time.

String constants are assigned to string variables with the LET statement, the INPUT statement, and the READ statement. (Refer to these keywords in the Character String section for details.)

Array Variables

Array variables are variables which represent an array of numbers. An array of numbers can have either one dimension or two dimensions.

One Dimensional Arrays

The following array is an example of a one dimensional array of numbers:

1	25	14	78	−Ø.35	1.89E+6

This array contains six elements. An element can be any real number expressed in either standard notation or scientific notation.

Two Dimensional Arrays

An array can also have two dimensions. For example, the following array is a two dimensional array:

Columns

1	**2**	**3**	
1	2	3	← **Row 1**
4	5	6	← **Row 2**
7	8	9	← **Row 3**

This two dimensional array has three rows and three columns. A two dimensional array is called a matrix.

Arrays can be assigned to any valid numeric variable symbol, however, the symbol must first be defined as an array variable in a DIM statement. For example, if the statement DIM F(1Ø) is executed, the variable F is defined to be a one dimensional array variable with a maximum working size of ten elements. If the statement DIM G(3,5) is executed, the variable G is defined to be a two dimensional array variable with a maximum working size of 15 elements. A DIM statement can appear anywhere in a BASIC program. The DIM statement is discussed in detail later in this section.

Once an array variable is dimensioned, it can be assigned elements via the LET statement, the INPUT statement, or the READ statement. Assigning elements to an array is discussed later in this section.

The BASIC interpreter inputs and outputs two dimensional arrays in row major order. For example, if the previous matrix is assigned to the array variable B5 and stored on magnetic tape with a PRINT @33:B5; command, the BASIC interpreter first outputs the elements in row one (1,2,3), followed by the elements in row two (4,5,6), followed by the elements in row three (7,8,9). The elements are stored on magnetic tape in a continuous string as follows: CR CR 1 2 3 CR 4 5 6 CR 7 8 9 CR CR. Notice that a Carriage Return character is automatically inserted as a delimiter as the end of each row.

Subscripting Array Variables

An entire array is referenced by referring to its assigned array variable, however, if you want to refer to a particular element in an array, you must use a subscripted array variable. For example, if the one dimensional array M has 1Ø elements, then you specify the sixth element in the array as M(6). The (6) is called a subscript.

The same rule applies to two dimensional arrays. If Q is an array variable representing a 3 by 3 matrix, then any reference to the variable Q refers to the entire matrix. If, however, you want to refer to the third row, and the second element in that row, you use the subscripted array variable Q(3,2). The first number in the subscript refers to the row, the second number refers to the column. For example:

$$Q = \begin{bmatrix} 1Ø & 2Ø & 3Ø \\ 4Ø & 5Ø & 6Ø \\ 7Ø & 8Ø & 9Ø \end{bmatrix}$$

After this matrix is entered into memory, the statement PRINT Q; prints the entire matrix on the GS display in two dimensional form as shown above. If, however, the statement PRINT Q(2,2) is executed, the BASIC interpreter prints the number 5Ø, because 5Ø is the element located in the second row, second column. It should be noted that zero and negative numbers are always invalid (out of range) subscripts.

Since subscripted array variables represent numeric constants, a subscripted array variable can be substituted for a numeric variable anywhere a numeric variable is specified in a syntax form; there are exceptions to this rule: the index to a FOR and NEXT statement must be a numeric variable, the variable used in a DEF FN statement and the target variables specified in the POLL statement must be numeric variables.

ARITHMETIC, LOGICAL, AND RELATIONAL OPERATORS

Operators are BASIC language elements which perform an operation on one or two parameters. A parameter can be specified as a constant, a variable or an expression. (Expressions are defined later in this section.) For example, the plus (+) operator adds two parameters together and the division operater (/) divides one parameter by another.

There are basically two types of operators; monadic and dyadic. Monadic operators require only one parameter. The plus sign (+) and the minus sign (−) in the numbers +5 and −3 are examples of monadic operators. Dyadic operators require two parameters. For example, the multiplication operator (*) and the division operator (/) are dyadic operators because two parameters are required to perform the operation. The operators in the Graphic System BASIC language are divided into the following categories:

Arithmetic Operators
Logical Operators
Relational Operators
The String Concatenation Operator
Array Operators
Scalar/Array Operators

Arithmetic Operators

There are seven arithmetic operators in the language which perform an arithmetic operation. The following table summarizes the results obtained from each arithmetic operator.

Operator	Description	Example	Result
↑	Exponentiation	3↑2	9
*	Multiplication	4*3	12
/	Division	12/4	3
+	Addition	5+2	7
−	Subtraction	6−5	1
MIN	Returns the smaller parameter	−3 MIN −4	−4
MAX	Returns the larger parameter	−3 MAX −4	−3

Logical Operators

There are three logical operators in the language: AND, OR, and NOT. These logical operators correspond to their boolean algebra equivalent. Two numbers are required as parameters with each operator. The operator returns a logical 1 or a logical Ø based on a comparison between

the two numbers. If the absolute value of a number is less than .5, the number is treated as a logical Ø. If the absolute value of a number is equal to or greater than .5, the number is treated as a logical 1. The following table summarizes the results obtained from each logical operator:

Operator	Description	Example	Result
AND	Returns the logical AND of the parameters	1 AND Ø	Ø
OR	Returns the logical OR of the parameters	1 OR Ø	1
NOT	Returns the logical NOT of the parameters	NOT 1	Ø

Relational Operators

Relational operators compare two parameters and return a logical result. A logical 1 is returned if the relationship is true. A logical Ø is returned if the relationship is false. There are six relational operators in the language: Equal (=), Not Equal (< >), Less Than (<), Greater Than (>), Equal To Or Greater Than (= >) and Equal To Or Less Than (= <). The following table summarizes the results returned to each relational operator:

Operator	Description	Example	Result
=	Returns the logical result	3 = 4	Ø
< >	Returns the logical result	3 < > 4	1
<	Returns the logical result	3 < 4	1
>	Returns the logical result	3 > 4	Ø
= >	Returns the logical result	3 = > 4	Ø
= <	Returns the logical result	3 = < 4	1

Relational operators can also be used to compare two character strings. The two character strings are compared character by character starting with the left most character in each string and proceeding to the right. The first difference determines the relationship. The characters are compared according to the priority established in the ASCII Character Priority Chart in Appendix B. Upper case letters are considered equal to lower case letters (i.e., BUGGS = buggs) unless the NOCASE environmental parameter is set. (Refer to the SET statement in the Environmental Control section for details.)

If one string ends before a difference is found, the shorter string is considered smaller in value.

The relational operators return a logical 1 if the relationship is logically true and a logical Ø if the relationship is logically false. For this reason, a relational comparison between two character

strings enclosed in parentheses can be specified as part of a numeric expression, because the relational comparison as a whole is reduced to a numeric constant. (Numeric expressions are discussed later in this section.)

The following table summarizes the results returned by relational operators when string constants are specified as parameters:

Operator	Description	Example	Result
=	Returns the logical result	("Bugs" = "Bunny")	Ø
<>	Returns the logical result	("Bugs" <> "Bunny")	1
<	Returns the logical result	("Bugs" < "Bunny")	1
>	Returns the logical result	("Bugs" > "Bunny")	Ø
=>	Returns the logical result	("Bugs" => "Bunny")	Ø
=<	Returns the logical result	("Bugs" =< "Bunny")	1

Refer to the IF . . . THEN . . . statement in the Controlling Program Flow section for more information or relational operators.

The String Concatenation Operator

The string concatenation operator (&) performs an operation which concatenates (joins) two character strings together. For example, if the BASIC interpreter evaluates the statement A$ = "BAT" & "MAN", the two string constants "BAT" and "MAN" are joined together to form the string constant "BATMAN" which is then assigned to the string variable A$. The string constant resulting from the concatenation process must be assigned to a string variable (A$ in this case).

Only two string constants can be concatenated in one statement. For example, A$ = B$ & C$ & D$ is not allowed. Also, the concatenation of two strings can not be specified as part of another statement. For example, the statement A$ = SEG (B$ & C$, 3, 2) is not allowed.

Array Operators

The following operations are allowed on numeric arrays. In any operation involving two arrays, both arrays must have the same dimension and the same number of elements. All variables shown in the examples below are array variables which have been previously dimensioned in a DIM statement.

Arithmetic (monadic) Operations

Operator	Description	Example
−	Changes the sign of all elements.	B = −A
+	No effect	B = +A

Arithmetic (dyadic) Operations

Operator	Description	Example
*	Element by element multiply	C = A*B
/	Element by element divide	C = A / B
↑	Element by element exponentiation	C = A ↑ B
+	Element by element add	C = A + B
−	Element by element subtract	C = A − B
MIN	Element by element compare	C = A MIN B
MAX	Element by element compare	C = A MAX B

In each of the above cases, the operation is performed with an element in array A and its corresponding element in array B; the result is assigned to the corresponding element is array C. Each of the above operations must be executed as an assignment statement.

Logical Comparisons

Operator	Description	Example
AND	Returns the logical result	C = A AND B
OR	Returns the logical result	C = A OR B
NOT	Changes 1's to Ø's and Ø's to 1's	C =NOT A

Relational Comparisons

Operator	Description	Example
=	Element by element compare	C = (A = B)
<>	Element by element compare	C = A <> B
<	Element by element compare	C = A < B
>	Element by element compare	C = A > B
=>	Element by element compare	C = A => B
=<	Element by element compare	C = A =< B

In each of the above cases, an element in array A is compared to its corresponding element in array B; the result (Ø or 1) is assigned to the corresponding element in array C.

Scalar/Array Operators

The Graphic System BASIC language allows a scalar (a single number) and an array to be specified as parameters in a math operation. The following tables summarize the results obtained when a scalar and an array are specified as parameters to an arithmetic, logical, or relational operator. Each variable shown in a table represents a numeric array.

Arithmetic Operations

Operator	Description	Example
*	Element by element multiply	C = A*5
/	Element by element divide	C = A/4
↑	Element by element exponentiation	C = A↑3
+	Element by element add	C = A+2
–	Element by element substract	C = A–1
MIN	Element by element compare	C =Ø MIN A
MAX	Element by element compare	C =Ø MAX A

In each of the above cases, the operation is performed with the scalar (numeric constant) and an element in array A; the result is assigned to the corresponding element in array C.

Logical Comparisons

Operator	Description	Example
AND	Returns the logical result	C = 1 AND A
OR	Returns the logical result	C = Ø OR A
NOT	Returns the logical result	C = NOT A

Relational Comparisons

Operator	Description	Example
=	Element by element compare	C = (45 = A)
<>	Element by element compare	C =45 <> A
<	Element by element compare	C =45 < A
>	Element by element compare	C = 45 > A
=>	Element by element compare	C = 45 => A
=<	Element by element compare	C = 45 =< A

In each of the above cases, an element in array A is compared to the scalar (numeric constant) and the result (Ø or 1) is assigned to the corresponding element in array C.

Comments on Operators

1) If you are multiplying a number by an integer, place the integer in the left operand position (i.e. use 5*3.28 rather than 3.28*5) and the statement execution time will be reduced by a factor of 3.
2) When dividing by 2, the statement execution time is reduced by a factor of four, if you express the problem as .5*X rather than X/2.

NUMERIC FUNCTIONS AND STRING FUNCTIONS

Numeric Functions

Numeric functions are special purpose mathematical operations which return a numeric result based on a parameter. For example, the SIN function requires an angle for a parameter and returns the sine of the angle. The angle can be specified in radians, degrees, or grads.

In most cases, the parameter of a numeric function can be specified as a numeric expression. If the parameter is specified as a numeric expression, the expression must be enclosed in parentheses; otherwise, the parentheses are optional. For example, if the BASIC interpreter evaluates the statement LET Y = SIN 3+4, the BASIC interpreter assumes 3 is the function's parameter and takes the sine of 3, adds 4, and assigns the result to the numeric variable Y. If parentheses are not used, as shown, the BASIC interpreter assumes the first element following a function is the parameter. If, however, the entire expression 3+4 is the parameter of the function, then the expression must be enclosed in parentheses as in the statement LET Y = SIN (3+4). When this statement is evaluated, the BASIC interpreter first adds 3 and 4 to get 7, takes the sine of 7, and assigns the result to the numeric variable Y. When listing a program, the BASIC interpreter always places parentheses around the parameter to make the listing easier to read.

If a numeric expression can be specified as a parameter, an array can also be specified as a parameter. For example:

```
100 LET B=SIN A
```

When this statement is executed, the BASIC interpreter computes the sine of each element in array A and assigns the result to the corresponding element in array B. In this case, the array variables A and B must be conformable; that is, both array variables must have the same dimensions.

String Functions

String functions are special purpose functions which manipulate character strings. For example, the SEG (Segment) function extracts a substring from the main body of a string. String functions, by definition, produce string constants as a result, just as numeric functions produce numeric constants; however, some functions listed under string functions, such as the LEN (Length) function, are actually numeric functions because they return a numeric result. These functions are listed under string functions, however, because their purpose is to manipulate character strings. String functions which return a numeric constant can be specified in a numeric expression. String functions that return a string constant cannot be part of a numeric expression.

The result of a string function must be assigned to a target variable. For example, when the BASIC interpreter evaluates the statement A$ = STR (5.2), the numeric constant 5.2 is converted to a string constant " 5.2" and is assigned to the string variable A$. In this case, A$ is the target variable.

NUMERIC EXPRESSIONS

A numeric expression is defined as any combination of numeric constants, numeric variables, array variables, subscripted array variables, numeric functions, or relational comparisons enclosed in parentheses joined together by arithmetic operators, logical operators, or relational operators in such a way that the expression as a whole can be reduced to a numeric constant. In addition, a numeric expression can be comprised of one or more smaller numeric expressions joined together by arithmetic, logical, or relational operators, as long as the expression as a whole can be reduced to a numeric constant.

Arithmetic Expressions

An arithmetic expression is an expression which falls under the category of a numeric expression. An arithmetic expression is defined as any combination of numeric constants, numeric variables, array variables, subscripted array variables, numeric functions, or relational comparisons (enclosed in parentheses), joined together by arithmetic operators in such a way that the expression as a whole can be reduced to a numeric constant. For example:

```
X↑2+3*X+5
```

This arithmetic expression contains the variable X and the numeric constants 2,3, and 5 joined together with the exponentiation operator (↑), the addition operator (+), and the multiplication operator (*). If the variable X is previously assigned the value 5, for example, and this arithmetic expression is entered in to the system from the GS keyboard and the RETURN key pressed, the BASIC interpreter replaces X with its assigned value (5) and evaluates the expression. In this case, the arithmetic expression reduces to the numeric constant 45. If X does not have an assigned value when the expression is evaluated, an undefined variable error occurs and the appropriate error message is printed on the GS display.

The BASIC interpreter follows normal math hierarchy when evaluating an arithmetic expression; exponentiation is performed first, followed by division and multiplication, followed by addition and subtraction. This execution order can be changed, however, by using parentheses. For example, if the result of the previous arithmetic expression is to be divided by 5, then the appropriate entry is as follows:

```
(X↑2+3*X+5)/5
```

In this case, the BASIC interpreter reduces the arithmetic expression inside the parentheses to the numeric constant 45, then divides 45 by 5 to get 9. If the parentheses were not used, the BASIC interpreter would perform the division (5/5) before adding the terms. The result would be 41 instead of 9.

Execution priority is discussed in detail later in this section.

Logical Expressions

Logical expressions are defined as any combination of numeric constants, numeric variables, subscripted array variables, numeric functions, numeric expressions, or relational comparisons (enclosed in parentheses) joined together by the logical operators AND, OR and NOT, in such a way that the expression as a whole can be reduced to a numeric constant (Ø or 1). For example:

```
X OR Y AND NOT A OR B OR ("DOG"="CAT")
```

When this logical expression is evaluated, the values assigned to the variables X,Y,A, and B are treated as a logical 1 or a logical Ø. All values equal to or greater than .5 are treated as a logical 1; values less than .5 are treated as a logical Ø. The string relational comparison ("DOG"="CAT") is reduced to a logical Ø because the string constants are not equal. In this case, if X=Ø, Y=1, A=1, and B=1, then the logical expression as a whole is reduced to a logical 1. This logical 1 can be treated as numeric data which allows this logical expression to be specified as part of a larger numeric expression.

Relational Expressions

Like arithmetic and logical expressions, relational expressions are considered a subset of the broad category of numeric expressions.

Relational expressions are defined as a combination of numeric constants, numeric variables, subscripted array variables, numeric functions, numeric expressions, or logical comparisons joined together by one or more relational operators in such a way that the expression as a whole can be reduced to a single numeric value (Ø or 1). For example:

```
("ZIG"<"MARK")<=("JERRY"<>"TERRY")>=("TAN">"TOO")
```

When this relational expression is evaluated, the string relational comparison ("ZIG"<"MARK") is reduced to a logical Ø, the string relational comparison ("JERRY"<>"TERRY") is reduced to a logical 1, and the string relational comparison ("TAN">"TOO") is reduced to a logical Ø. These results are then compared as follows:

```
Ø<=1>=Ø
```

This comparison returns a logical 1. This result can be treated as numeric data which allows this expression to be specified as part of a larger numeric expression.

Complex Numeric Expressions

Any combination of arithmetic expressions, logical expressions, and relational expressions can be joined together by arithmetic, logical, and relational operators to form a numeric expression, as long as, the expression as a whole can be reduced to a numeric constant. And, any number of numeric expressions can be joined together by arithmetic, logical, and relational operators as long as the expression as a whole can be reduced to a numeric constant. Careful use of parentheses is required in most cases to keep the operations straight. For example, the following numeric expression is a valid numeric expression and can be specified as a parameter to a keyword, if the syntax form of the keyword states that numeric expressions are allowed.

```
45+(X↑3+2*SIN(X)-3)↑Ø.5+("BEAR"="HARE" OR A$<B$)+5Ø*RND (-2)+Y↑2/3
```

In this case, the variables X, A$, B$, and Y must have assigned values before the numeric expression is evaluated.

This is an extreme example to be sure, and it doesn't have much practical value, but it does emphasize the tremendous degree of freedom one has to build a numeric expression to the point where it solves very complex and unusual problems.

Execution Priority

The following table lists the execution priority followed by the BASIC interpreter when a BASIC statement is executed. The operator with the highest priority is labeled number 1.

PRIORITY	OPERATOR
1	Left Paren (
2	Functions
3	Monadic operators Plus (+), Minus (−), and NOT
4	Exponentiation operator (↑)
5	Arithmetic operators Multiplication (*) and Division (/)
6	Arithmetic operators Addition (+) and Subtraction (−)
7	Arithmetic operators MIN and MAX
8	Relational operators: =, < >, <, >, = <, = >
9	The logical operators AND and OR
10	The keyword USING and Comma (,)
11	Right Paren) and semicolon (;)
12	The keywords OF, THEN, STEP, TO, and the symbols @, #, %, and = (the assignment operator)
13	All other keywords
14	Carriage Return

NUMERIC ERRORS

Fatal Errors

The term "fatal error" refers to any error that causes program execution to terminate. Normally, math operations with invalid parameters or math operations which produce out of range numbers generate fatal errors.

Size Errors

A SIZE error occurs when a math operation produces an out of range number. For example, when the function EXP (71Ø) is evaluated, the BASIC interpreter attempts to raise the base e (the natural logarithm base) to the power 71Ø. The result is an out of range number (a number outside the range ±1.ØE+3Ø8). The BASIC interpreter returns the largest number it can (8.988465674E+3Ø7) and generates a SIZE error message. This error condition is treated as a fatal error and program execution is terminated, unless an ON SIZE THEN . . . statement has been previously executed in the BASIC program. (Refer to the Handling Interrupts section for details.) Normally, the result returned by the BASIC interpreter when a SIZE error occurs can be predicted. If the result of a math operation exceeds the upper boundary of the numeric range, the number +8.988465674E+3Ø7 is returned; this number is defined to be plus infinity for the system (+ ∞). If the result of a math operation exceeds the lower boundary of the numeric range, the number −8.988465674E+3Ø7 is returned; this number is defined to be minus infinity for the system (− ∞). If the result of a math operation is a small number which approaches Ø and falls within the range ±1.ØE−3Ø8, a SIZE error is not generated. If the result of a math operation is closer to zero than 1.112536929E−3Ø8, Ø is returned as the result.

The following table lists math operations which produce predictable out of range numbers. In some math operations, like the tangent of 9Ø degrees, the BASIC interpreter treats the SIZE error as though an error didn't occur and program exeuction continues on its normal path. In other cases, the operation produces the results as shown and a SIZE error condition is set. In these cases, an ON SIZE THEN... statement must be in the BASIC program to handle the error condition or the error is treated as a fatal error and program execution is aborted.

Numeric Error Conditions				
MATH OPERATION	CAUSE OF ERROR	EXAMPLE	NUMBER RETURNED	ERROR TYPE
Addition (+) Subtraction (–) Multiplication (*) Division (/)	Parameter too Large or too Small	1E2ØØ*1E2ØØ	+ ∞	SIZE
		–1E2ØØ*1E2ØØ	– ∞	SIZE
		1/1E2ØØ	Ø	NO ERROR
	Division by Zero	4/Ø	+ ∞	SIZE
		– 4/Ø	– ∞	SIZE
Exponentiation (↑)	Parameter too Large or too Small	1E2ØØ↑1E2ØØ	+ ∞	SIZE
		–1E2ØØ↑1E2ØØ	– ∞	SIZE
	A<Ø and B not an integer in the range Ø to 255	– 2 ↑ 3	– 8	NO ERROR
		– 2 ↑ 6.5	+ ∞	FATAL
Square Root	Negative parameter	SQR(–4)	2	SIZE
Sine X	| X | ≥ 4.116E+5 (radians)	SIN (4.2E+5)	Ø	SIZE
Cosine X	| X | ≥ 4.116E+5 (radians)	COS (4.2E+5)	Ø	SIZE
Tangent X	| X | ≥ 4.116E+5 (radians)	TAN (4.2E+5)	Ø	SIZE
TAN 9Ø°	Parameter Out of Range	SET DEG TAN (9Ø)	– ∞	NO ERROR
		SET DEG TAN (–9Ø)	+ ∞	NO ERROR
e^x	Parameter Out of Range	EXP (71Ø)	+ ∞	SIZE
		EXP (–71Ø)	Ø	SIZE
Matrix Inversion	Determinant is Ø	INV X	Undetermined Answer	SIZE
Matrix Multiply	Floating Point Overflow	A MPY B	Answers + ∞ – ∞	SIZE

THE DIM STATEMENT

Syntax Form:

```
[ Line number ] DIM { string variable ( numeric expression )
                    { array variable ( numeric expression [ , numeric expression ] ) }

    [ , { string variable (numeric expression)
        { array variable (numeric expression) [ , numeric expression ] ) } ] ...
```

Descriptive Form:

```
[ Line number ] DIM { string variable ( maximum number of characters)
                    { array variable ( first dimension [ , second dimension ] ) }

    [ , { string variable ( maximum number of characters )
        { array variable ( first dimension [ , second dimension ] ) } ] ...
```

Purpose

The DIM (Dimension) statement is used to reserve memory space for one or more string variables and/or one or more array variables.

Explanation

Dimensioning String Variables

If a character string is assigned to a string variable without dimensioning the string variable first, the maximum working size of the string variable is automatically dimensioned to 72 characters by default. String variables are therefore dimensioned with the DIM statement for two reasons; to make the maximum working size larger than 72 characters, or to make the maximum working size smaller than 72 characters.

Increasing the Maximum Working Size. If a character string is assigned to a string variable from the GS keyboard, the default working size of 72 characters is adequate because keyboard entries are limited to 72 characters by the size of the line buffer. If, however, the string variable receives the results of a string concatenation operation or is specified as the target to receive a character string from a peripheral device, the incoming character string might contain more

than 72 characters; if so, the string variable must first be dimensioned to a larger size or an error occurs. For example:

```
200 DIM A$(200),B$(500)
```

When this statement is executed, 200 bytes of memory space are reserved for A$ and 500 bytes of memory are reserved for B$. This allows up to 200 characters to be assigned to A$ and up to 500 characters to be assigned to B$.

The working size of a string variable can be dimensioned as large as the memory capacity of the system allows. Remember, however, that memory space reserved in this manner is taken away from the space used to store the BASIC program; this means that less space is available to store BASIC statements.

Once a string variable is dimensioned, either by the DIM statement or by default when the string assignment is made, the working size can be reduced with another DIM statement, but it can not be increased unless the variable is first deleted with the DELETE statement. For example; assume that the working size of A$ is to be reduced by 100 characters and the working size of B$ is to be increased by 100 characters. The appropriate statements are as follows:

```
210 DELETE B$
220 DIM A$(100),B$(600)
```

When line 210 is executed, the space reserved for B$ returns to an unreserved status. The 200 bytes reserved for A$ remain reserved. When line 220 is executed, the maximum working size for A$ is reduced to 100 characters and 600 bytes of memory are reallocated to B$ for a working space. Although the working size of A$ is reduced by 100 characters, 200 bytes are still reserved for A$ because the variable was not deleted first. This means that 100 bytes of memory are not available for assignment. At a later time, however A$ can be redimensioned back to 200 characters without deleting the variable first. As a general rule, a variable can be redimensioned to any size less than its original maximum working size, but never greater than its maximum working size without deleting it first. Once a string variable is dimensioned, a character string can be assigned to the variable with the LET statement, the INPUT statement, or the READ statement. Refer to the LET statement in this section or the INPUT and the READ statements in the Character String section for details.

Dimensioning Array Variables

Array variables must be dimensioned before they can be assigned a value. Any valid numeric variable symbol can be used as an array variable symbol as long as the symbol does not have an

assigned value. (Refer to the topic VARIABLES at the beginning of this section for a list of the available numeric variable symbols.) Once a numeric variable symbol is dimensioned as an array variable, it can no longer be used to represent a scalar—a numeric constant, unless the array is first deleted from memory. Once deleted, the symbol can be reused to represent a numeric constant.

The following statement illustrates how an array variable is dimensioned:

```
100 DIM A(10),B(5,5)
```

When this statement is executed under program control, the variable A is dimensioned to be a one-dimensional array with a working size of ten elements. Enough space is reserved in memory to store one numeric value for each element, in this case approximately 80 bytes (8 bytes per element). This storage space looks like this:

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

Attaching a subscript to the variable symbol produces a symbol which represents an element in the array. In this case, A(5) refers to the fifth element in array A. An array variable is like a string variable; once its working size is established, the variable can be redimensioned to a smaller working size, but never to a larger working size without deleting it first. In this case, A can be redimensioned to a five element array, a seven element array, or even a 2x5 two dimensional array without deleting the variable first.

The second variable in statement 100 is dimensioned to be a two dimensional array (a matrix) with five rows and five columns. The space reserved in memory looks something like this:

B(1,1)	B(1,2)	B(1,3)	B(1,4)	B(1,5)
B(2,1)	B(2,2)	B(2,3)	B(2,4)	B(2,5)
B(3,1)	B(3,2)	B(3,3)	B(3,4)	B(3,5)
B(4,1)	B(4,2)	B(4,3)	B(4,4)	B(4,5)
B(5,1)	B(5,2)	B(5,3)	B(5,4)	B(5,5)

Each element in array B is referred to by attaching a subscript as shown in the illustration. For example, B(2,2) refers to the element in the second row, second column. Enough space is reserved in memory to store one numeric value for each element. In this case, space for 25 numeric constants is reserved.

Like a one dimensional array, a two dimensional array can be dimensioned to a smaller size without deleting the variable first, but never to a larger size. This array variable can be redimensioned to a 4x5 matrix, a 2x2 matrix, and even a 1x25 matrix without deleting the variable first. But to increase the number of elements in the array, the variable B must be deleted and then redimensioned.

Notes on Dimensioning Variables

Any number of string variables and array variables can be dimensioned in the same DIM statement, as long as the statement does not exceed 72 characters or the amount of memory required does not exceed the available memory space. The working size of a variable can be specified as a numeric expression as long as the BASIC interpreter can reduce the expression to a numeric constant and round the constant to an integer within the range 1 through 65530 for string variables, and the range 1 through 8191 for array variables. Again, the capacity to the random access memory actually limits the maximum working size of a variable. (Refer to the Memory Management section for information on how to estimate the memory space taken up by an array.)

Once an array variable is dimensioned, the variable symbol refers to the entire array; the variable symbol with a subscript refers to one element in the array. For example:

```
250 PRINT B
260 PRINT B(2,2)
```

If the variable B is dimensioned as an array and each element is assigned a value, then line 250 above causes the BASIC interpreter to print the entire array on the GS display. Line 260 causes the BASIC interpreter to print the element in the second row, second column.

Once an array variable is dimensioned, each element can be assigned a value with the LET statement, the INPUT statement, or the READ statement. (Refer to the LET statement in this section and the INPUT and READ statements in the Input/Output Operations section for details.

THE LET STATEMENT

Syntax Form:

[Line number] [LET] { array variable = numeric expression / string variable = string expression / numeric variable = numeric expression }

Descriptive Form:

[Line number] [LET] { array variable = numeric expression / string variable = string expression / numeric variable = numeric expression }

Purpose

The LET statement is used to assign values to variables as a BASIC program executes.

Explanation

The LET statement requires a variable as a parameter, followed by the assignment operator (=), followed by an expression which represents the value to be assigned to the variable. For example:

```
100 LET Y=X12+2* X+3
```

The variable to the left of the assignment operator (=) can be any valid variable symbol; either a numeric variable, an array variable, a subscripted array variable, or a string variable. Multiple assignment (i.e. A=B=2) is not allowed.

If the variable to the left of the assignment operator is a numeric variable or a subscripted array variable, then the assigned value on the right of the equal sign can be a numeric expression, a numeric function, a numeric variable, or a numeric constant. The assigned value can be almost anything as long as the BASIC interpreter can evaluate and reduce the numeric expression to a numeric constant. If variables are included in the numeric expression, the variables must have assigned values by the time the statement is executed, or an error occurs.

If the variable to the left of the assignment operator is an array variable, the array variable must be previously dimensioned with the DIM statement.

If the variable to the left of the assignment operator is a string variable, the assigned value must be a string constant enclosed in quotation marks, another string variable with an assigned value, a string function, or two string constants or string variables joined together with the concatenation operator (&).

The Keyword LET is Optional

The keyword LET is an optional entry which can be left out or it can be included as part of the statement for clarity and documentation purposes. For example:

```
11Ø LET Y=X↑2+2*X+3
12Ø Y=X↑2+2* X+3
```

These two statements are identical as far as the BASIC interpreter is concerned. In a program listing, however, it's a little more obvious that line 11Ø is an assignment statement, at least at first glance.

Assigning Values to Numeric Variables

A numeric variable is assigned a value in the following way:

```
LET X=−3.25
```

When this statement is entered from the GS keyboard and the RETURN key is pressed, the BASIC interpreter executes the statement immediately, because the statement doesn't have a line number. The numeric constant −3.25 is assigned to the variable X. The minus monadic operator (−) is considered part of the number. Immediate results may not be seen at first, but pressing the X key followed by pressing the RETURN key causes the BASIC interpreter to print the value of X on the GS display. (This technique can be used anytime to examine the contents of a variable.)

If the assignment statement is preceded by a line number, the BASIC interpreter stores the statement in memory as part of the current BASIC program. When the RUN statement is entered from the GS keyboard and the RETURN key is pressed, the assignment is made when the statement is executed under program control.

The value assigned to a numeric variable can be a numeric constant, a numeric variable, a numeric function, or a numeric expression as long as the BASIC interpreter can reduce the

entry to a numeric constant when the statement is evaluated. The following statements are examples of the different kinds of assignments that can be made:

```
130 LET X=X+3
140 LET Y=SIN(2*PI*X+5)
150 C=(A↑2+B↑2)↑.5
```

In line 130, the numeric constant 3 is added to the current value of X and the total is reassigned to X. In line 140, the BASIC interpreter evaluates the numeric expression inside the parentheses, treats the result as an angle expressed in the current trigonometric units for the system, and assigns the sine of the angle to the numeric variable Y. And in line 150 the BASIC interpreter reduces the numeric expression on the right side of the assignment operator (=) to a numeric constant and assigns the result to the numeric variable C. In each case, the variables on the right side of the assignment operator must have assigned values by the time the statement is evaluated, or an error occurs and program execution is aborted.

Assigning Character Strings to String Variables

Character strings (string constants) are assigned to string variables in the following manner:

```
160 LET A$="I'm the Graphic System at Your Service !"
```

When this statement is executed under program control, the character string on the right side of the assignment operator (=) is assigned to the string variable A$. The enclosing quotation marks act as string delimiters to mark the beginning and end of the string. (These quotation marks are not considered to be a part of the string.)

Any string variable from A$ to Z$ can be selected as the target to receive the character string. If the string variable already has an assigned value, that value is overwritten by the new character string when the assignment is made. If the character string on the right side of the assignment operator is larger than 72 characters, the string variable on the left side of the operator must be dimensioned to a size large enough to accommodate the character string before the assignment is made. If the character string is less than or equal to 72 characters, then the string variable is automatically dimensioned to 72 characters when the assignment is made (unless the string variable was previously dimensioned to a smaller size with the DIM statement). Refer to the DIM statement in this section for details.

The assignment statement can be used to assign the results of a string function or the results of a string concatenation operation to a string variable. For example:

```
170 LET M$=SEG(A$,X,13)
180 LET W$=M$&R$
```

When line 17Ø is executed, a substring in the character string assigned to A$ is assigned to M$. The number assigned to the variable X specifies the starting location of the substring and the substring contains 13 character. (Refer to the SEG function in the Character String section for details on the SEG function.)

When line 18Ø is executed, the string constant assigned to M$ is concatenated (joined) to the end of the character string assigned to M$. The result is assigned to the string variable W$. Care must be taken when executing a concatenation operation that the string variable on the left of the assignment operator is dimensioned large enough to accommodate the resultant character string.

Assigning Numeric Values to Array Variables

A numeric array entered into memory must always be represented by an array variable. The first step in assigning an array to an array variable is to select an undefined (unused) numeric variable symbol and dimension the variable in a DIM statement. (This procedure is discussed under the DIM statement in this section.) After the variable is dimensioned, the elements in the array are assigned numeric values using subscripts on the array variable. For example, assume the following array is to be entered into memory:

```
1Ø   2Ø
3Ø   4Ø
```

Selecting B as a variable symbol, the first step is to dimension B with the DIM statement as follows:

```
DIM B(2,2)
```

This statement sets the working size of B to two rows and two columns, and enough space is reserved in memory to store four numeric values; one value for each element.

The next step is to assign a numeric constant to each element. If the assignment statement is used, each element must be assigned a value using subscripts on the array variable as follows:

```
B(1,1) = 1Ø
B(1,2) = 2Ø
B(2,1) = 3Ø
B(2,2) = 4Ø
```

The above statements can be entered from the GS keyboard as shown for immediate assignment, or they can be entered with line numbers for assignment under program control. Either way, the result is as follows:

$$B = \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix}$$

After each element has an assigned value, the entire array can be examined by pressing the B key and the RETURN key on the GS keyboard. This causes the BASIC interpreter to print the array on the GS display.

The BASIC interpreter also allows the elements in an array to be assigned the same value in one assignment statement. For example:

```
100 DIM A(10),B(2,2)C(5,5)
110 LET A = 0
120 LET B = 5
130 LET C = SIN(X)
```

In line 110 all of the elements in array A are made equal to zero (0). In line 120, all of the elements in array B are made equal to 5. And, in line 130 all of the elements in array C are made equal to the sine of X. (In this case, the variable X must have an assigned value or an error occurs.)

Arithmetic operations can be performed on arrays as part of an assignment statement. For example:

```
100 LET M = SQR N
```

In this statement, the BASIC interpreter takes the square root of each element in array N and assigns the result to the corresponding element in array M. Both array M and N must be conformable; that is they must have the same dimensions. Specifically,

$$\text{If } N = \begin{bmatrix} 9 & 25 \\ 16 & 36 \end{bmatrix} \quad \text{Then } M = \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix}$$

An array variable can be specified as the parameter to any numeric function. The function performs the indicated operation on each element in the array and assigns the result to the specified target array.

Making Assignments with the INPUT and READ statements

Another way to assign values to the elements of an array is to use the INPUT statement as follows:

```
INIT
DIM B(2,2)
INPUT B
```

When these statements are executed from the GS keyboard, the variable B is dimensioned to a 2x2 matrix. The INPUT statement then places a blinking question mark on the GS display and the BASIC interpreter prepares to assign keyboard entries to the elements of array B. The elements must be entered in row major order as follows:

```
? 10,20,30,40
```

Each entry can be separated by a comma or by pressing the RETURN key. The BASIC interpreter keeps displaying the blinking question mark until an entry is made for each element in the array. The last entry must be followed by pressing the RETURN key, then the blinking question mark goes away.

The READ statement can also be used to assign values to the elements in an array. For example:

```
200 INIT
210 DIM B(2,2)
220 DATA 10,20,30,40
230 READ B
```

When these statements are executed, the numeric constants in the DATA statement (line 220) are assigned to array B in row major order.

(Refer to the Input/Output Operations section of this manual for complete information on the READ statement and the INPUT statement.)

A Note about the Equality Relational Operator (=)

If the equality relational operator (=) is used in a BASIC statement, care must be taken not to confuse it with the assignment operator. For example, the assigned value of the variable A can be compared to the numeric constant 5 in two ways:

```
5 = A
 or
A = 5
```

In the first statement, the BASIC interpreter compares the numeric constant 5 to the value of A and returns a logical one if they are equal; a logical zero if they are not equal. In the second statement, the BASIC interpreter assumes an assignment operation is taking place and assigns the numeric constant 5 to the numeric variable A. If this is not intended, the results can be undesirable because the original value of A is overwritten with 5.

If it's obvious the statement is a relational comparison, the BASIC interpreter can figure it out. For example, when the statement IF A=B THEN 200 is executed, a relational comparison of A to B is obvious and the BASIC interpreter knows it should compare A to B rather than assign the value of B to the variable A.

ENVIRONMENTAL CONTROL

Section 2

ENVIRONMENTAL CONTROL

INTRODUCTION TO ENVIRONMENTAL CONTROL

The statements SET, INIT, FUZZ, and PRINT provide the facility to change the operating environment of the system under program control. The programmable environmental conditions are the DEGREE/RADIAN/GRAD setting, the TRACE/NORMAL setting, the KEY/NOKEY setting, the CASE/NOCASE setting and the FUZZ standards for comparing a number with Ø or two numbers with each other. In addition, two processor status bytes and one internal magnetic tape status byte can be accessed via the PRINT statement using special primary and secondary addresses. This allows one of several courses of action to be specified on a PAGE FULL condition, alternate delimiters can be specified for PRINT, LIST, SAVE and INPUT operations, and the internal magnetic tape can be set up to read different magnetic tape formats. Different alphanumeric fonts can also be selected and alphanumeric scale and rotation information can be sent to external peripheral devices.

Setting Environmental Parameters

The SET statement allows the following environmental parameters to be set directly from the GS keyboard or set while the system is operating under program control:

Degree/Radian/Grad

This environmental parameter establishes the trigonometric units of measure for the system.

Trace/Normal

When this environmental parameter is set to TRACE, the BASIC interpreter prints the line number of each BASIC statement on the GS display before the statement is executed under program control. This feature allows the system operator to monitor program flow during program test and debugging sessions. Setting the parameter to NORMAL returns the system to normal operation.

Key/Nokey

Setting this parameter to KEY allows the BASIC interpreter to respond to the user-definable keys on the GS keyboard while the system is operating under program control. If a user-definable key is pressed while a BASIC program is executing, the BASIC program halts while the user-definable function is executed; the BASIC program then continues normal execution at the interruption point. Setting this parameter to NOKEY prevents the BASIC interpreter from responding to the user-definable keys while the system is operating under program control.

Case/Nocase

When this parameter is set to CASE, lower case letters are considered equal to upper case letters when relational comparisons are made between two character strings (i.e., "A"="a"). When this parameter is set to NOCASE, lower case letters are not considered equal to upper case letters.

Initializing the System

The INIT statement sets most of the programmable environmental conditions to a predefined state. The INIT statement provides a quick and easy method to re-establish the system environment to a known state from an unknown set of conditions.

Fuzzy Comparisions

The FUZZ statement sets the standard the BASIC interpreter uses when it compares a number with Ø and when it compares two numbers with each other. The concept of a "fuzzy comparison" is explained fully under the explanation section of the keyword.

The FONT Statement

This environmental parameter selects the character font used by the Graphic System display. Once this parameter is set, the only way to change it is to execute another FONT statement, or turn off the system power. The INIT statement has no effect on this parameter.

The "ALPHAROTATE" Parameter

The "ALPHAROTATE" parameter is used to send alphanumeric rotation information to an external peripheral device such as an X-Y plotter. A special PRINT statement is used to send the information.

The "ALPHASCALE" Parameter

The "ALPHASCALE" parameter is used to send alphanumeric scale information to an external peripheral device such as an X-Y plotter. A special PRINT statement is used to send the information.

The "PAGE FULL" Parameter

The "PAGE FULL" parameter allows a course of action to be specified when a page full condition occurs on the GS display. This environmental parameter can be set to any one of the following courses of action:

Blinking "F" in upper-left corner
Execute a HOME statement
Execute a PAGE statement
Execute a COPY statement, then a PAGE statement

Large-Screen Display Parameters

For the 4054 Graphic System, character size and display characteristics can be selected with the CHARSIZE and BRIGHTNESS statements.

The CHARSIZE Statement

This statement is used to select the size of characters on the 4054 Graphic System display. The INIT statement has no effect on this parameter.

The BRIGHTNESS Statement

This statement sets the intensity (BRIGHT/NORMAL) and focus (FOCUSED/DEFOCUSED) characteristics of the 4054 Graphic System display. The INIT statement has no effect on this parameter.

Specifying Alternate Delimiters for PRINT and INPUT Operations

Two processor status bytes can be accessed with a special PRINT statement to establish the delimiters used during PRINT, SAVE, LIST, and INPUT operations. Normally the BASIC interpreter uses the Carriage Return character to terminate each character string during ASCII output operations. This delimiter can be changed to Carriage Return/Line Feed with an environmental setting.

On INPUT operations, the BASIC interpreter treats Carriage Return as the logical record separator and hexidecimal FF as the End Of File mark. Any ASCII character can be specified as an alternate record separator and an alternate EOF character by changing an environmental parameter. In addition, the BASIC interpreter can be directed to delete an ASCII character each time it is found in the incoming ASCII data string.

The Magnetic Tape Status Parameters

The internal magnetic tape status byte can be accessed by a special PRINT statement and changed to allow the internal magnetic tape unit to read and write different magnetic tape formats. Three status parameters allow the internal magnetic tape unit to read and write with a 128 or 256 byte physical records, using the checksum error checking technique or without using the checksum error checking technique, using a "header" format or without using a "header" format.

THE "ALPHAROTATE" PARAMETER

Purpose

The "ALPHAROTATE" parameter sends alphanumeric rotation information to an external peripheral device on the General Purpose Interface Bus.

Explanation

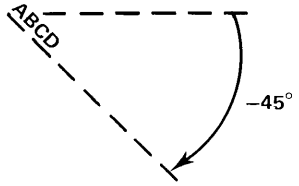
The Graphic System has the ability to send alphanumeric rotation information to an external peripheral device on the General Purpose Interface Bus (GPIB). The peripheral device receiving the information must have the facility to rotate alphanumeric characters accordingly. The GS display does not have this facility.

Alphanumeric rotation information is sent to an external peripheral device through a special PRINT statement. For example:

```
SET DEG
PRINT @16,25:−45
```

The statement SET DEG sets the trigonometric units for the system to degrees. The special PRINT statement then sends the alphanumeric rotation information to peripheral device 16 on the GPIB. The I/O address @16,25: is sent first. Primary address 16 tells device 16 to prepare to take part in an I/O operation. Secondary address 25 tells device 16 that the BASIC interpreter is about to send alphanumeric rotation information in the form of an ASCII character string.

The rotation angle is specified after the colon (:) in the special PRINT statement. The BASIC interpreter converts this information into an ASCII character string and sends the string to the specified peripheral device, in this case, device 16. It is up to device 16 to receive the ASCII string and set its internal rotation parameter to −45 degrees. The results are shown below:



Since this is an environmental command, an immediate result may not be seen until alphanumeric characters are printed by the receiving device. When the characters are printed, they are printed at a −45° angle to the horizontal as shown in the diagram. All characters printed by this device are printed at this angle until the "ALPHAROTATE" parameter is changed.

Actually, anything can be specified after the colon in the PRINT statement; it doesn't have to be the sine and cosine of the rotation angle. The key to setting the "ALPHAROTATE" parameter is the secondary address 25. This address tells the peripheral device to treat the ASCII character string as alphanumeric rotation information. The specified ASCII character string, whatever it is, must have meaning to the peripheral device and it is up to the peripheral device to set the rotation angle accordingly.

THE "ALPHASCALE" PARAMETER

Purpose

The "ALPHASCALE" parameter sends alphanumeric scale information to an external peripheral device on the General Purpose Interface Bus.

Explanation

The Graphic System has the ability to send Alphanumeric scale information to an external peripheral device on the General Purpose Interface Bus (GPIB). The peripheral device receiving the information must have the ability to interpret the information as alphanumeric scale information and set its internal scale parameters accordingly. The alphanumeric scale information is sent via a special PRINT statement. For example:

```
PRINT @16,17:X,Y
```

When this statement is executed, the I/O address @16,17: is sent over the GPIB. Primary address 16 tells peripheral device number 16 that it has been selected to take part in an I/O operation. Secondary address 17 tells peripheral device 16 that the information it is about to receive is alphanumeric scale information. The BASIC interpreter then converts the data items which follow the colon in the PRINT statement into an ASCII character string, and sends the character string to the specified peripheral device. In this case, the numeric value assigned to the variable X is sent first, followed by the numeric value assigned to the variable Y. Device 16 receives the ASCII string and interprets the first value as the horizontal scale factor; the second numeric value is assumed to be the vertical scale factor.

Actually, any type of data can be specified after the colon in the PRINT statement as long as the information can be interpreted by the receiving peripheral device as alphanumeric scale information. The key item in this PRINT statement is the secondary address 17. This secondary address tells the peripheral device to treat the ASCII data as alphanumeric scale information and to set its internal scale parameters accordingly.

THE BRIGHTNESS STATEMENT

Syntax Form:

[Line number] BRI numeric expression

Discriptive Form:

[Line number] BRIGHTNESS display code

NOTE

This command is not available in the 4051 and 4052 Graphic Systems.

Purpose

The BRIGHTNESS statement defines environmental parameters for the display.

Explanation

The BRIGHTNESS statement specifies the intensity and focus parameters for the display. The display code definitions are as follows:

Display Code	Intensity	Focus
0	Normal	Defocused
1	Normal	Focused
2	Bright	Defocused
3	Bright	Focused

Bright intensity increases the displayed intensity of characters and vectors; defocused lines appear wider than focused lines.

NOTE

The actual appearance of bright and defocused vectors may depend on internal adjustments to your Graphic System's display.

The default setting is 1 (normal, focused). This parameter is not reset by an INIT command.

The default address is PRINT @ 32,30:

THE CHARSIZE STATEMENT

Syntax Form:

[Line number] CHA numeric expression

Descriptive Form:

[Line number] CHARSIZE size code

NOTE

This command is not available in the 4051 and 4052 Graphic Systems.

Purpose

The CHARSIZE statement specifies the size of characters on the Graphic System display.

Explanation

Four character sizes are available for the 4054 Graphic System. The sizes and their effect on screen layout are shown here:

Size Code	No. of Characters/Line	No. of Lines/Page
1	132	64
2	119	58
3	79	38
4	72	35

The default size code is 4 (the largest characters).

NOTE

The CHARSIZE command only affects the size of characters on the display. BASIC statements and line editing are still limited to 72 characters (the length of the line buffer). To alter character size when plotting on a Tektronix 4660 Series Plotter, use the "ALPHASCALE" parameter.

The default address for the CHARSIZE command is PRINT @ 32,17:

THE FONT STATEMENT

Syntax Form:

[Line number] FON numeric expression

Descriptive Form:

[Line number] FONT font code

Purpose

The "ALPHAFONT" parameter selects one of six character fonts for the Graphic System display or an external peripheral device on the General Purpose Interface Bus.

Explanation

GS Display

The following character fonts can be selected for character output on the Graphic System Display:

FONT TYPE	FONT CODE
ASCII Font	Ø
Scandinavian Font	1
German Font	2
General European Font (French, British, Italian)	3
Spanish Font	4
Graphic Symbols Font	5
Business	8
Danish	9

The character font is automatically set to U.S. Font on system power up. After system power up, any one of the other character fonts can be selected by executing the following statement:

FONT font code

When this statement is executed, the font code is converted to an ASCII character string and sent to the GS dsiplay. The GS display then switches to the character font specified by the code. The font code can be a constant, a variable, or an expression.

NOTE

The keyword "FONT" is not available on the 4051 Graphic System. The special PRINT statement:

PRINT @32,18:font code

selects the desired font, as shown in the following table.

4051 Graphic System Fonts

	[{	SHIFT [{	] }	SHIFT] }	SHIFT # 3	SHIFT $ 4	\ |	SHIFT \ |	` @	
U.S.	Ç	ü	é	â	ê	ô	à	ç	ù	PRINT @ 32 , 18:0
Scandinavian	ä	Ä	å	Å	£	ø	ö	Ö	§	PRINT @ 32 , 18:1
German	ä	Ä	ü	Ü	£	ß	ö	Ö	§	PRINT @ 32 , 18:2
General European	Ç	ü	é	â	£	ê	à	ç	§	PRINT @ 32 , 18:3
Spanish	¡	ü	¿	â	£	ê	Ñ	ç	§	PRINT @ 32 , 18:4
Graphic	Ç	ü	é	â	▒	ê	à	ç	§	PRINT @ 32 , 18:5
ASCII Decimal Equivalent	91	123	93	125	35	36	92	124	64	

The following table shows you the changed symbols for each font on the 4052 and 4054 Graphic Systems.

	SHIFT #3	ø	@	[	N	]	SHIFT [	SHIFT N	SHIFT]	
ASCII	#	ø	@	[	^	]	{	|	}	FONT 0
SWEDISH	#	ø	@	ä	ö	å	Ä	Ö	Å	FONT 1
GERMAN	£	ø	@	ä	ö	ü	Ä	Ö	Ü	FONT 2
BRITISH	£	ø	@	[	^	]	{	|	}	FONT 3
SPANISH	#	ø	@	¡	Ñ	¿	{	|	}	FONT 4
GRAPHIC	#	ø	§	[	^	]	÷	¼	÷	FONT 5
RESERVED	Same as FONT 0									FONT 6
RESERVED	Same as FONT 0									FONT 7
BUSINESS	£	ø	@	[	^	]	{	|	}	FONT 8
DANISH	#	ø	@	Æ	ø	Å	æ	ø	å	FONT 9

External Peripheral Devices

If an external peripheral has the capability to change character fonts, the alphafont information can be sent to that device over the General Purpose Interface Bus (GPIB). For example:

```
PRINT @16,18: 5
```

When this statement is executed, the I/O address @16,18: is sent over the GPIB. Primary address 16 tells peripheral device number 16 that it has been selected to take part in an I/O operation. Secondary address 18 tells device 16 that the BASIC interpreter is about to send alphafont information in the form of an ASCII character string. The number 5 is then converted into an ASCII character string and sent to device 16. It is up to device 16 to receive the character string and interpret the number 5 as a font code. Actually, anything can be specified after the colon in the PRINT statement as long as it is a valid numeric value or a valid character string. It must, however, have meaning to the specified peripheral device.

THE FUZZ STATEMENT

Syntax Form:

[Line number] FUZ numeric expression [, numeric expression]

Descriptive Form:

[Line number] FUZZ number of digits for comparisons not involving zero

[, numeric value of closeness for comparisons with zero]

Purpose

The FUZZ statement sets the standard used by the BASIC interpreter when two non-zero numbers are compared with each other or when a number is compared with absolute zero.

Explanation

The Graphic System does not compute mathematical operations with infinite precision; therefore, it is necessary to provide a facility which sets the standard for comparing two numbers which are extremely close to each other. The FUZZ statement provides that facility.

Comparing Two Non-Zero Numbers

Should the BASIC interpreter consider the number 4.ØØØØØØØØØ1 equal to 4.Ø when making a comparison or shouldn't it? This is an example of a "fuzzy" comparison; both numbers are extremely close to each other. The deciding factor for this comparison is the first parameter set by the FUZZ statement.

Assume the following statements are executed under program control:

```
100 INIT
110 FUZZ 10
120 X=4.00000000001
130 Y=4.0
140 IF X=Y THEN 200
 .
 .
 .
```

When line 100 is executed, the system environmental parameters are reset to their initial state. Line 110 sets the comparison standard for comparing two non-zero numbers to 10 digits. Numeric assignments are then made to the variables X and Y in lines 120 and 130, and the numbers are compared in line 140. In this case, the comparison standard is set to 10 digits and the first 10 digits of both numbers are identical, so the branch to line 200 occurs.

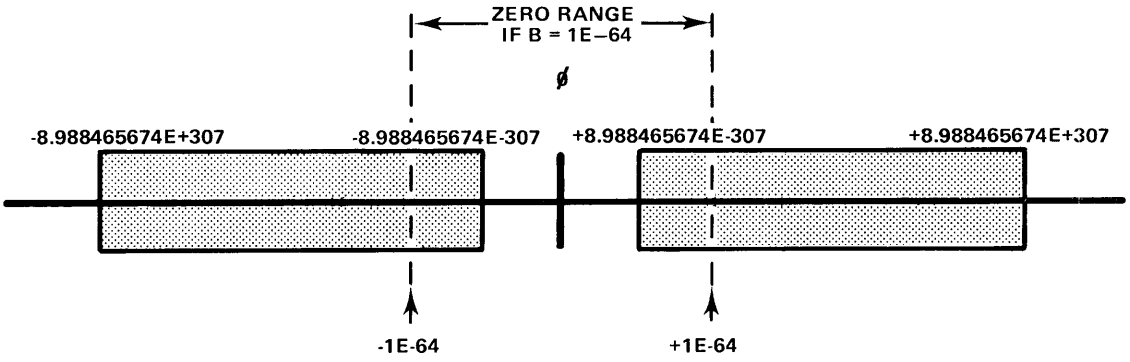
If line 110 is changed to FUZZ 11 instead of FUZZ 10, then the branch doesn't occur because the eleventh digits of the two numbers are not equal.

In this example, the first parameter of the FUZZ statement is specified as a numeric constant (10). This parameter can be specified as a numeric expression as long as the BASIC interpreter can reduce the numeric expression to a numeric constant and round the constant to a positive integer.

This parameter is automatically set to 12 on system power up and after the execution of an INIT statement.

Comparing a Number with Absolute Zero

The following diagram is a graphic representation of the numeric range for the system:



The horizontal line represents the real number world: the line extends in both directions from absolute zero (Ø). All positive numbers are located to the right of zero; all negative numbers are to the left of zero. The shaded portions of the line represent the numeric range of the system. Due to hardware limitations, the largest positive number allowed is +8.988465674E+ 3Ø7 as shown in the diagram on the right; the smallest positive number is + 8.988465674E−3Ø7. The largest negative number is −8.988465674E+ 3Ø7; the smallest negative number is −8.988465674E−3Ø7. Notice there is a gap between the smallest positive number and the absolute value of zero, and the smallest negative number and the absolute value of zero. Are the numbers −8.988465674E−3Ø7 and + 8.988465674E−3Ø7 equal to zero for all practical purposes, or aren't they? The second parmeter specified in the FUZZ statement is the deciding factor.

Assume that the following series of statements are executed:

```
31Ø LET A = 1Ø
32Ø LET B = 1E−64
33Ø FUZZ A,B
34Ø LET X = −1E−2ØØ
35Ø IF X=Ø THEN 6ØØ
```

In line 33Ø, the comparison standard for comparing a number with zero is set to 1E−64 (the assigned value of B). This range is shown in the diagram. With the standard set to this value, all positive numbers equal to or less than +1E−64 are considered equal to zero and all negative numbers equal to or greater than −1E−64 are considered equal to zero. Therefore, when line 35Ø is executed, the assigned value of X is considered equal to zero and the branch to line 6ØØ occurs. If the value of X changes to −1E−1Ø for example, then the branch to line 6ØØ doesn't occur, because −1E−1Ø isn't within the zero range set by FUZZ.

The standard for comparisons with zero can be set to any positive numeric value within the range of the system. For example, if line 32Ø is LET B = 1, then the BASIC interpreter considers all positive numbers equal to or less than +1 to be equal to zero, and all negative numbers equal to or greater than −1 to be equal to zero. If Ø itself is specified as the standard, then only absolute zero is considered equal to zero.

The standard for comparisons with zero can be specified as a numeric expression as long as the BASIC interpreter can reduce the entry to a positive numeric constant when the FUZZ statement is executed.

THE INIT STATEMENT

Syntax Form:

[Line number] INI

Descriptive Form:

[Line number] INIT

Purpose

The INIT statement returns most of the programmable environmental parameters for the system to a known state.

Explanation

When an INIT statement is executed, either directly from the GS keyboard or under program control, the following environmental parameters are set as follows:

1. All variables enter an undefined state.
2. All system interrupt functions previously activated with an ON. . . THEN. . . statement are inactivated.
3. The IFC (Interface Clear) signal on the General Purpose Interface Bus is activated to set all interface circuitry to a known quiescent state.
4. All files are closed.
5. The DATA statement pointer is restored to the first data item in the first DATA statement.
6. All DEF FN functions are returned to an undefined state.
7. The parameters of FUZZ are set to 12 and 1E—64; that is, 12 digits for non-zero comparisons and 1E—64 for comparisons with Ø.
8. The trigonometric units selection RADIAN/DEGREE/GRAD is set to RADIAN.
9. The TRACE/NORMAL debugging feature is set to NORMAL.
10. The KEY/NOKEY interrupt facility is set to NOKEY.

12. The CASE/NOCASE comparison feature is set to CASE (i.e., upper case letters are equal to lower case letters).
13. The WINDOW parameters are set to Ø,13Ø,Ø,1ØØ.
14. The VIEWPORT parameters are set to Ø,13Ø,Ø,1ØØ.
15. The SCALE parameters are set to 1,1.
16. The ROTATE parameter is set to Ø.
17. For the 4054 Graphic System, the dash mask is set to Ø.

These environmental conditions are also established on system power up. Refer to the SET statement in this section for more information on the environmental parameters RADIAN/DEGREE/GRAD, TRACE/NORMAL, KEY/NOKEY, and CASE/NOCASE.

THE INTERNAL MAGNETIC TAPE STATUS PARAMETERS

The status byte for the Graphic System internal magnetic tape unit can be changed so that different magnetic tape formats can be read. The status byte is addressed via a special PRINT statement as follows:

Statement	Meaning
PRINT @33,Ø:Ø,Ø,Ø	256 byte physical record, checksum, header format
PRINT @33,Ø:1,1,1	128 byte physical record, no checksum, non-header format

The I/O address @33,Ø: in the PRINT statement selects the internal magnetic tape status byte as the target to receive the parameter changes. Three numbers are specified as parameters after the I/O address. The first parameter specifies the physical record length; Ø selects a 256 byte physical length, 1 selects a 128 byte length. The second parameter specifies whether the magnetic tape unit uses the checksum error checking technique or whether it doesn't; Ø selects checksum, 1 selects no checksum. The third parameter specifies whether the magnetic tape unit uses a file header format or a non-header format; Ø selects header format; 1 selects a non-header format. Any combination of 1's and Ø's can be specified for these three parameters.

Physical Record Length

When files are created on magnetic tape with the MARK statement, each file is divided into a number of physical records of equal size. The number of bytes in each record is controlled by the first parameter in the magnetic tape status byte. If the first parameter is set to Ø, then the file is created with 256 byte physical records. If the first parameter is set to 1, then the file is created with 128 byte physical records. For example:

```
100 FIND Ø
110 PRINT @33,Ø:Ø,Ø,Ø
120 MARK 1,1ØØØ
```

When line 100 is executed, the tape head is positioned to the beginning of the magnetic tape. Line 110 sets the internal magnetic tape status parameters to 256 byte physical record, checksum, and header format. Line 120 then creates one new file on the magnetic tape. In this case, five physical records are created on tape with 256 bytes of storage space per record. The first physical record is reserved for the file header because the third status parameter is Ø. The next four physical records are reserved for storing data. Enough space is created to hold the specified number of bytes (1000) while keeping the number of physical records to a whole number. (This must be done because partial physical records can not be created.)

If line 11Ø is changed to PRINT @33,Ø:1,Ø,Ø then the file is created using 128 byte physical records. In this case, the first physical record is reserved for the file header. Eight physical records are then created to serve as the file storage area.

With the first magnetic tape parameter set to 1, only a magnetic tape with a 128 byte physical record format can be read. When the first parameter is set to Ø, only 256 byte physical records can be read.

Checksum

The term "checksum" refers to an error checking technique used by the Graphic System when magnetic tapes are recorded and read. When a tape is recorded using checksum, all of the data bits in a physical record are added up; the total is recorded as the last data byte in the record. When the tape is read, the data bits are counted. If the total doesn't match the number recorded in the last byte in the record, then a read error is assumed to have occurred.

The checksum error checking technique is used unless the second parameter of the status byte is set to 1. Setting this parameter to 1 is necessary to read tapes which have not been recorded using checksum. When 1 is specified as the second parameter, the number of bytes in each physical record are counted instead of the number of bits. The total must be 128 or 256, whichever is specified by the first status byte parameter, or the BASIC interpreter assumes a read error has occurred.

Header Format

If the third parameter of the magnetic tape status byte is set to 1, the BASIC interpreter assumes the first physical record of the file marks the beginning of the file storage area. The BASIC interpreter also assumes the data in the file is stored in ASCII format. Therefore, READ and WRITE operations to and from a binary data file cannot be performed with the status byte set to non-header format.

If an ASCII file is marked in header format and the magnetic tape status byte is set to non-header format, then the information stored in the file header is input as the first logical record when an INPUT operation is performed on the file. (Refer to the FIND statement for more information on accessing a tape file header.)

Magnetic Tape Format Compatibility

Changing the internal magnetic tape status byte gives the Graphic System the ability to read magnetic tapes recorded on other magnetic tape recording equipment. For example, executing the statement PRINT @33,Ø:1,1,1 gives the Graphic System the ability to read and write in a magnetic tape format which is compatible with the format used by the Tektronix 4923 Digital Cartridge Tape Recorder.

Resetting the Status Byte

The magnetic tape status byte can be reset to its initial state (Ø,Ø,Ø) by pressing the AUTO LOAD key on the GS keyboard, by executing a PRINT @33,Ø:Ø,Ø,Ø statement, or by turning off the system power.

NOTE

Storing data on the magnetic tape using different status parameters other than those set when the tape was MARKed may cause the loss of some files. Different tape formats should not be mixed on the same tape.

THE PAGE FULL PARAMETER

One of several courses of action can be specified when a PAGE FULL condition occurs on the GS display during program execution. A PAGE FULL condition occurs whenever the display cursor moves off the bottom of the screen. When this happens, the BASIC interpreter takes one of the following courses of action:

Page Full Parameter Setting	Action Taken During Program Execution
PRINT @32,26: Ø	Displays a Blinking "F" in upper-left corner.
PRINT @32,26: 1	Return the Cursor to the HOME Position.
PRINT @32,26: 2	Execute a PAGE command.
PRINT @32,26: 3	Execute a MAKE COPY command then a PAGE command.

Blinking "F"

If the statement PRINT @32,26:Ø is executed from the GS keyboard or under program control, a blinking "F" is displayed whenever a PAGE FULL condition occurs. The PAGE FULL selection is set automatically to the blinking "F" on system power up.

NOTE

When the screen is only echoing input data from the keyboard, the PAGE FULL parameters have no effect.

HOME

If the statement PRINT @32,26:1 is executed from the GS keyboard or under program control, a HOME command is automatically executed when a PAGE FULL condition occurs, if the system is operating under program control.

PAGE

If the statement PRINT @32,26:2 is executed from the GS keyboard or under program control, then a PAGE command is automatically executed when a PAGE FULL condition occurs, if the system is operating under program control.

MAKE COPY and PAGE

If the statement PRINT @32,26:3 is executed from the GS keyboard or under program control, a MAKE COPY command, then a PAGE command is automatically executed each time a PAGE FULL condition occurs, if the system is operating under program control.

THE PROCESSOR STATUS PARAMETERS

Four environmental parameters controlling input and output delimiters are set by addressing two processor status bytes via a special PRINT statement. One status byte controls the input and output delimiters used during INPUT, OLD, APPEND, PRINT, LIST, and SAVE operations. This status byte is addressed by specifying I/O address @37,26: in the special PRINT statement. The second status byte determines the alternate delimiters used on INPUT operations when a percent sign (%) is specified in the INPUT I/O address instead of an "at" sign (@). This status byte is addressed by specifying @37,Ø: in the special PRINT statement.

Changing the ASCII Input /Output Delimiter from CR to CR/LF

The Graphic System normally uses CR (Carriage Return) as the input and output delimiter for all data transfers in ASCII code. This delimiter can be changed, however, by executing the following statement, either directly from the GS keyboard or under program control:

```
PRINT @37,26:1
```

When this statement is executed, primary address 37 selects the processor status bytes as the target to receive the parameter change information. Secondary address 26 selects the status byte which controls the delimiter for ASCII I/O operations. The 1 following the colon tells the processor to use CR/LF as an output delimiter instead of CR. The 1 also tells the processor to delimit every incoming ASCII data string on a LF (Line Feed) character instead of the CR (Carriage Return) character.

This status byte is returned to its initial power up state by executing the following statement:

```
PRINT @37,26:Ø
```

The Ø in this statement tells the processor to use CR instead of CR/LF as a delimiter in ASCII data transfers.

Selecting Alternate Delimiters for INPUT, OLD, and APPEND Operations

If a percent sign (%) is specified in place of the "at" sign (@) in the I/O address for the INPUT, OLD, or APPEND statement, the BASIC interpreter uses a previously specified ASCII character for a record separator character and a previously specified ASCII character for an End Of File mark. This feature gives the Graphic System the ability to adapt its ASCII input format requirements to the ASCII output formats used by different peripheral devices. The ASCII characters to be used as the alternate record separator and End Of File mark are specified in a special PRINT statement.

The alternate delimiters and character to be deleted are selected by addressing the second processor status byte as follows:

PRINT @37,Ø:Ø–255,Ø–255,Ø–255

When this statement is executed (either directly from the GS keyboard or under program control) the alternate delimiters are established. Primary address 37 tells the processor to prepare to receive information which represents a change in a processor status byte. Secondary address Ø tells the processor that the parameters to the second status byte are to be changed. Three numbers separated by commas are then specified after the colon (:) in the PRINT statement. These numbers each represent the decimal equivalent of an ASCII character.

The first number after the colon represents the decimal equivalent of the record separator character to be used in the INPUT operation and must be in the range Ø–255. For example, if 65 is specified, the ASCII letter "A" is used as the record separator instead of CR.

The second number specified after the colon in the PRINT statement represents the End of File (EOF) character to be used and must be in the range Ø–255. For example, if 66 is specified as the second number, the first "B" found in the incoming ASCII data string is treated as an EOF mark. When a "B" is found, program execution is terminated and an EOF error message is printed on the GS display.

The third number specified after the colon indicates which ASCII character is to be deleted from the incoming ASCII data string. For example, if 67 is specified, the ASCII character "C" is deleted from the ASCII data string each time it appears. Again this number represents the decimal equivalent of an ASCII character. If the number specified as the third parameter is greater than 127, then every character is retained in the incoming data string.

Once these status byte parameters are set, the only way they can be changed is to execute another PRINT @37,Ø: statement or turn off the system power.

The decimal equivalents of ASCII characters can be found in Appendix B.

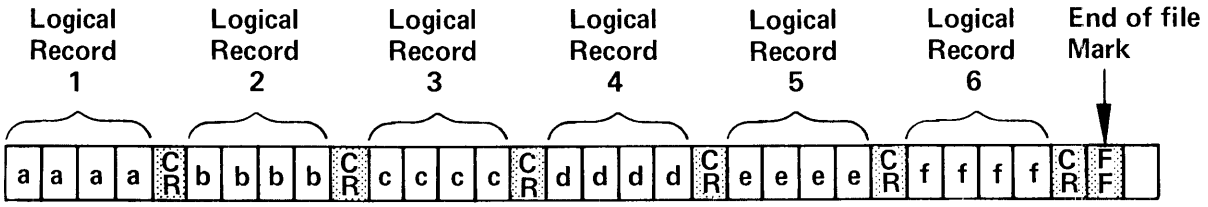
NOTE

Alternate delimiters do not affect the Graphic System keyboard. Input from the keyboard always uses Carriage Return for the Record Separator.

Specifying an Alternate Record Separator.
Care must be taken when specifying an alternate record separator delimiter to ensure that parts of logical records are not lost. This can happen if the ASCII data string contains both Carriage Return characters and the alternate record separator character. The following examples illustrate how this happens:

Example 1—Normal Delimiting Act on for INPUT operations.

5ØØ INPUT @2Ø:A$,B$,C$,D$,E$,F$,G$



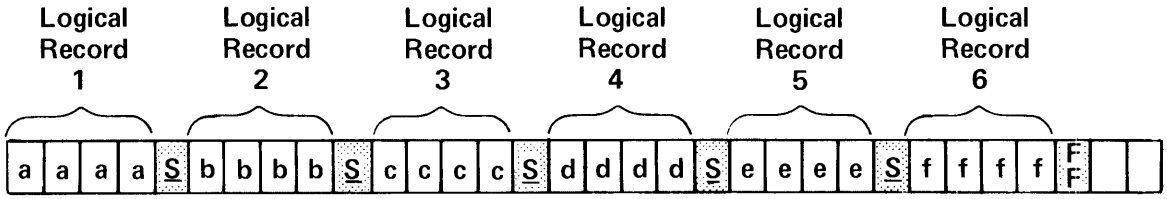
This example illustrates how normal delimiting action occurs during an input operation. If the "at" sign (@) is specified, Carraige Return is the only valid record separator character and hexidecimal FF is the only valid End Of File character. The character strings shown above represent ASCII character strings received from peripheral device number 2Ø when statement 5ØØ is executed. Each CR character marks the end of a logical record. The BASIC interpreter re-addresses peripheral device number 2Ø after each CR is received to tell it to send the next logical record. The logical record assignments are made as follows:

A$="aaaa"
B$="bbbb"
C$="cccc"
D$="dddd"
E$="eeee"
F$="ffff"
G$=End Of File Mark

When an attempt is made to assign the End of File mark (hexidecimal FF) to G$, program execution stops and the appropriate message is printed on the GS display. This is the same as executing a STOP statement.

Example 2—Delimiting Action when the Alternate Record Separator is specified.

```
51Ø PRINT @37,Ø:19,255,255
52Ø INPUT %2Ø:A$,B$,C$,D$,E$,F$,G$,
```



When line 51Ø is executed, the alternate record separator character is specified as ASCII decimal equivalent 19 (the DC3 control character). The End Of File character is specified as hexidecimal FF (255) and the character to be deleted from the ASCII data string is specified as 255 (no character to be deleted). Notice that when the processor status parameters are set, all three parameters must be specified, regardless of whether they are changed from their previous values or not.

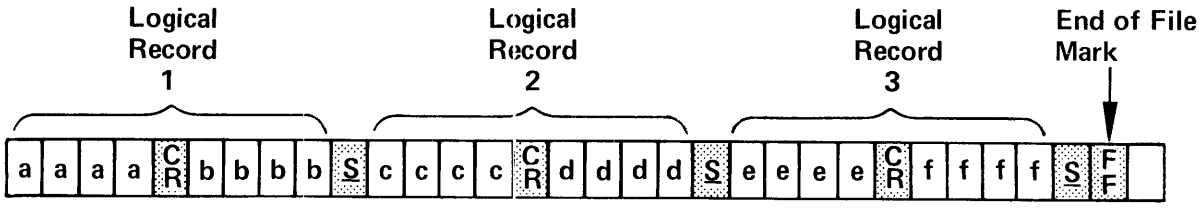
Line 52Ø inputs logical records from peripheral device 2Ø on the General Purpose Interface Bus. Because the percent sign (%) is specified in the I/O address instead of the "at" sign (@), the BASIC interpreter considers the DC3 control character (represented by the S symbol)as a valid record separator. The ASCII data string received from peripheral device 2Ø is shown above. The following assignments are made:

```
A$="aaaa"
B$="bbbb"
C$="cccc"
D$="dddd"
E$="eeee"
F$="ffff"
G$=End Of File Mark
```

Notice that the logical record assignments are the same as the previous example. Each time a DC3 control character is found, the BASIC interpreter treats the character as the end of a logical record. The BASIC interpreter re-addresses peripheral device number 2Ø after each DC3 is found and tells it to send the next logical record. This happens six times until an attempt is made to input the End of File mark and assign it to G$. In this case, peripheral device number 2Ø uses the same End Of File character as the normal value (hexidecimal FF). When this character is received, program execution stops and the appropriate message is printed on the GS display.

Example 3—Intermixing Carriage Returns and the Alternate Record Separator Character.

```
53Ø PRINT @37,Ø:19,255,255
54Ø INPUT %2Ø:A$,B$,C$,D$,E$,F$,G$
```



This example shows what happens when the Carriage Return character and the alternate Record Separator character are alternately used between logical records. The same program lines are re-executed that were used in the last example. This time, however, peripheral device number 2Ø sends the ASCII data strings shown in the illustration. Because the percent sign is specified in the INPUT statement, the BASIC interpreter assumes that only three records are sent over the bus, each terminated with the DC3 control character as shown in the illustration. The following assignments are made:

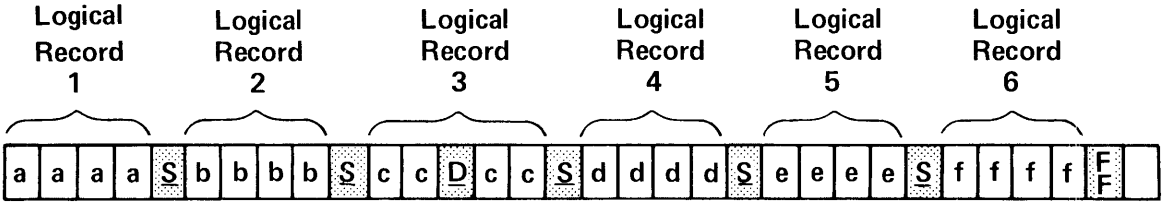
A$="aaaa"
B$="cccc"
C$="eeee"
D$=End Of File Mark

Because the Graphic System does not have the capability to handle the CR character as part of a character string coming in via an INPUT statement, the string assignment for each logical record is terminated at the CR character. In this example, the characters "bbbb", "dddd", and "ffff" are lost from the ASCII data string.

Specifying an Alternate End of File Character.

The second parameter in the PRINT statement specifies the alternate End Of File character. This parameter is specified as a decimal number between Ø and 255 and represents the decimal equivalent of an ASCII character. For example:

```
55Ø PRINT @37,Ø:19,4,255
56Ø INPUT %2Ø:A$,B$,C$,D$,F$,G$
```



In line 55Ø, the alternate record separator character is specified as decimal 19 (the DC3 control character) and the alternate End Of File character is specified as decimal 4 (the EOT control character). In line 56Ø, peripheral device 2Ø sends the ASCII string shown in the illustration. The assignments are made as follows:

A$="aaaa"
B$="bbbb"
C$="cc"

The first two logical records are assigned to A$ and B$, respectively. Notice that the alternate record separator (S) is used. When an attempt is made to assign the third logical record to C$, the alternate End Of File character is found. The characters in the third logical record up to the D character are assigned to C$. An error occurs when an attempt is made to input the D character, program execution is aborted, and the appropriate end of file error message is printed on the GS display.

NOTE

To handle an EOF interrupt condition with an ON... THEN... statement, a logical file number must be specified. An alternate EOF character will not be detected by an ON EOF(n) THEN... statement unless the input operation is on logical file number n.

Specifying a Character to be Removed From the ASCII Data String.

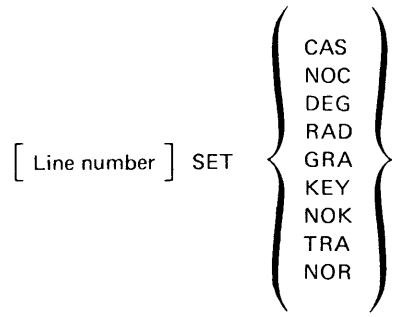
The third parameter specified in the PRINT statement tells the BASIC interpreter which character to remove from the incoming ASCII data string. If the parameter is specified as an integer from 1 to 127, the BASIC interpreter assumes the integer is the decimal equivalent of an ASCII character and removes that character each time it appears in the ASCII data string. If this parameter is specified as an integer greater than 127, but less than 256, then all of the characters are retained in the incoming ASCII data string.

Specifying the Processor Status Parameters as Numeric Expressions.

Each of the three processor status parameters can be specified as numeric expressions and set under program control as long as each numeric expression can be reduced to a numeric constant between Ø and 255. Any numeric constant outside this range results in an error, program execution is aborted, and a system error message is printed on the GS display.

THE SET STATEMENT

Syntax Form:



Descriptive Form:

[Line number] SET environmental condition

Purpose

The SET statement is used to set the trigonmetric units for the system, the trace debugging feature, the interrrupt facility for the user-definable keys, and the standard for comparing upper and lower case letters.

Explanation

Setting the Trigonometric Units

Executing a SET statement for RADIAN, DEGREE, or GRAD establishes the units of measure for trigonometric operations. For example:

```
100 SET DEGREES
110 X=SIN (45)
120 SET RADIANS
130 Y=SIN(45)
140 SET GRADS
150 Z=SIN(45)
```

When line 1ØØ is executed, the trigonometric units are set to degrees. Line 11Ø then assigns the sine of 45 degrees to the variable X. Line 12Ø sets the trigonometric units to radians and line 13Ø assigns the sine 45 radians to the variable Y. Line 14Ø is executed next and the trigonometric units are set to grads (one grad equals 1/1ØØ of a right angle). Line 15Ø assigns the sine of 45 grads to the variable Z. It can be seen in this example that even though SIN(45) is assigned to each variable, the results are different because the trigonometric units are different in each case. Care must be taken that the trigonometric units are set properly for trigonometric operations.

Setting the Trace Debugging Feature

Setting TRACE causes the BASIC interpreter to print the line number of a statement before it is executed. The line number is printed starting at the present position of the cursor; the BASIC interpreter then executes a carriage return (CR), then executes the statement. Normally, if the program does not involve graphic statements, the line numbers are printed in a single column on the left side of the GS display. When the screen is full, program execution halts until a PAGE is executed to erase the screen. After the screen is erased, program execution continues.

The TRACE feature allows you to monitor the execution order of the current program; it is a valuable aid in finding the source of run-time errors which occur as special conditions are set up during program execution. Program execution can be monitored by setting TRACE, then executing RUN; or TRACE can be SET, then the program can be executed one statement at a time by pressing the STEP PROGRAM key on the GS keyboard.

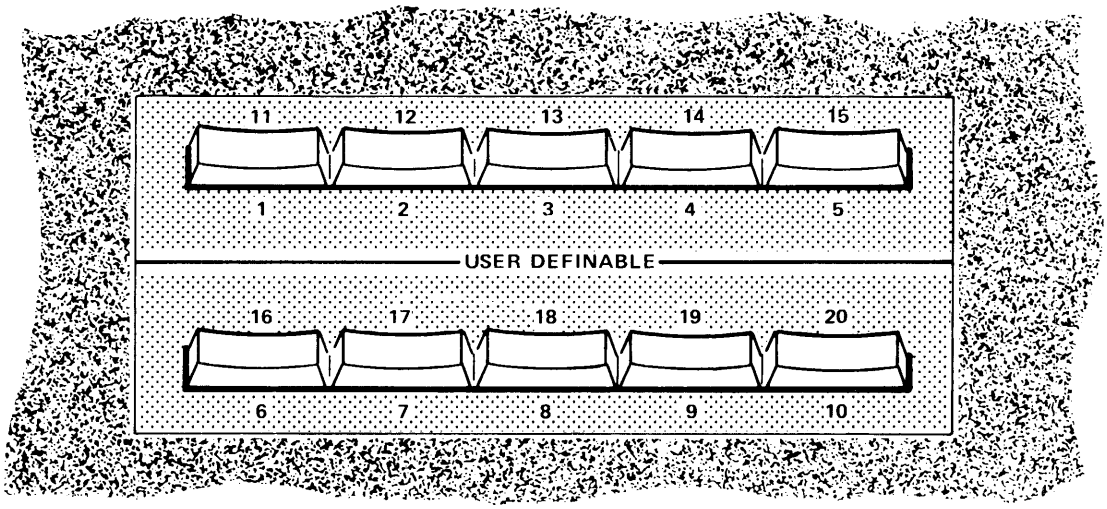
To disable the TRACE feature, the statement SET NORMAL must be executed from the GS keyboard or under program control. This feature is automatically set to NORMAL on system power up and after the execution of an INIT statement.

Activating User-Definable Key Interrupts

Executing the statement SET KEY allows the BASIC interpreter to respond to the user-definable keys while a BASIC program is executing. If a user-definable key is pressed while a program is executing, program execution halts after the current instruction is completed; the BASIC interpreter executes the user-definable function then returns to the interruption point in the BASIC program and resumes normal program execution. If a user definable key is pressed while a program is executing a keyboard INPUT statement, the INPUT statement is terminated, the values that have been entered are assigned to the target variables and the BASIC interpreter executes the user-definable function. After the user-definable function is completed the BASIC interpreter returns to the line following the INPUT statement.

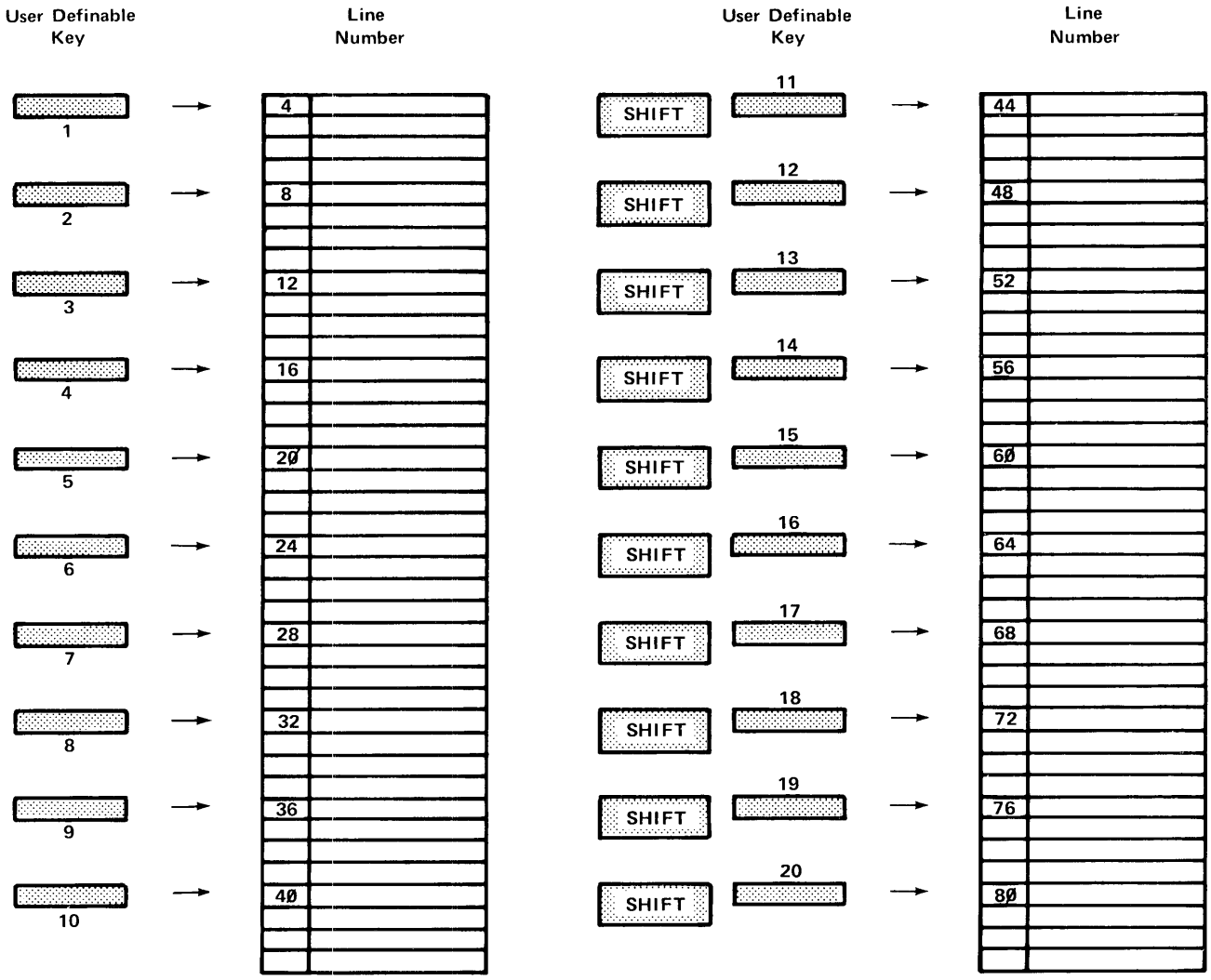
If the statement SET NOKEY is executed, this feature is disabled and the BASIC interpreter can not respond to the user-definable keys while the system is operating under program control. This feature is set to NOKEY (No Key) on system power up and after the execution of an INIT statement.

User-Definable Key Operation. At this point it might be appropriate to review the operation of the user-definable keys on the Graphic System keyboard. There are ten user-definable keys on the GS keyboard as shown in the following diagram:



Each key represents two user-definable functions. For example, if the key in the upper left-hand corner is pressed, user-definable function number 1 is executed; if the same key is pressed in combination with the SHIFT key, user-definable function number 11 is executed.

Each user-definable function is actually a program subroutine located in memory. Pressing a user-definable key is the same as executing a GOSUB (Go to Subroutine) statement; program control is transferred to the line number which is four times the key number. For example, key number 1 transfers program control to line number 4; key number 2 transfers program control to line number 8; key number 3 transfers program control to line number 12, and so on. The following diagram shows the line numbers associated with each user-definable key. These line numbers are fixed and cannot be changed.

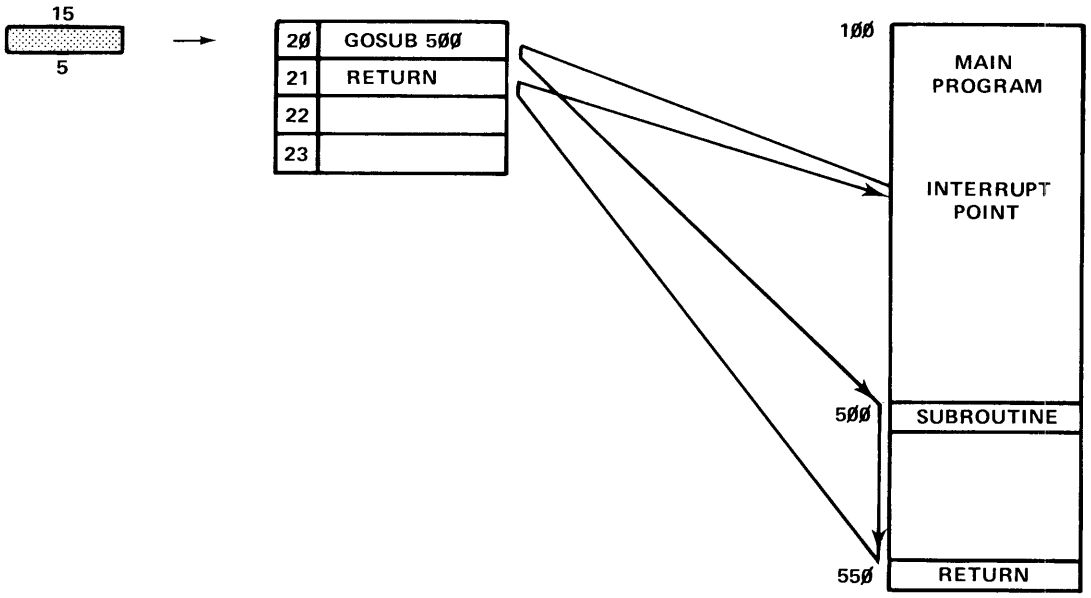


If key number 5 is pressed, for example, and the system is operating under program control, and the statement SET KEY has been executed, then the BASIC interpreter transfers program control to line number 20 after the current instruction is executed. If the system is idle when the key is pressed, the system is placed under program control and line 20 is executed as the first instruction.

After line 2Ø is executed, lines 21, 22, and 23 are executed in sequential order; if line 23 is not an END, STOP, or RETURN statement, then program control is passed on to the next user-definable function (line 24) and the BASIC interpreter keeps executing statements in sequential order until an END, STOP, or RETURN statement is found or until the BREAK key is pressed.

If a user-definable function ends in a RETURN statement, program control is transferred back to the interruption point in the main program, if the system was previously operating under program control. If the system was in an idle state when the key was pressed, the RETURN statement returns the system back to an idle condition.

Sometimes more than four statements are required for the subroutine. In this case, a GOSUB statement can be used to transfer program control to a larger subroutine. For example:



This diagram illustrates how a user-definable key can be used to transfer program control to a large subroutine in memory. When user-definable key number 5 is pressed, program control is transferred from the main BASIC program to line number 2Ø. Line number 20, in turn, transfers control to line number 5ØØ, the beginning of a larger subroutine. When the subroutine is finished executing, the RETURN statement at the end of the subroutine transfers control back

to the statement following the GOSUB statement, in this case line number 21. Line number 21 is also a RETURN statement and transfers program control back to the interruption point in the main program. The main program then continues normal sequencial execution just like nothing ever happened.

If the system is idle when key number 5 is pressed, the system is placed under program control and line number 2Ø is executed as the first instruction. Control is passed to line number 5ØØ and the subroutine is executed. The RETURN statement at the end of the subroutine transfers control back to line 21 just as it did before. This time however, a different course of action is taken. Because the system was not operating under program control when the key was pressed, the RETURN statement in line 21 returns the system back to keyboard control.

It is good practice to always start numbering the lines in a BASIC program with line number 1ØØ. This keeps the main BASIC program out of the area reserved for the user-definable keys. If a RENUMBER statement is executed, the renumbering operation automatically starts with line number 1ØØ, unless a lower line number is specified as the third parameter.

Upper and Lower Case Letter Comparisons

When the statement SET CASE is executed, the BASIC interpreter considers lower case letters equal to upper case letters when making relational comparisons between character strings. For example:

```
16Ø SET CASE
17Ø IF "RABBIT"="rabbit" THEN 2ØØ
```

When line 16Ø is executed, the BASIC interpreter is instructed to treat lower case letters equal to upper case letters. A string relational comparison is then made in line 17Ø. Since the only difference between the two character strings is the fact that one is upper case and one is lower case, the strings are considered equal and relationship is true. The branch to line 2ØØ is executed. If statement 16Ø is changed to SET NOCASE, then the two character strings would not be considered equal and the branch wouldn't occur. The CASE/NOCASE parameter is automatically set to CASE on system power up and after the execution of an INIT statement.

SYSTEM CONTROL

Section 3

SYSTEM CONTROL

INTRODUCTION TO SYSTEM CONTROL

The statements discussed in this section causes system control functions to be executed. These control functions include calling a specialized firmware routine, sending a make copy command to a Hard Copy Unit, returning the alphanumeric cursor to the home position on the GS display, and paging the GS display screen.

Calling a Specialized Firmware Routine

Firmware routines are special processor instructions which are permanently fixed in a memory chip and mounted in a plastic housing called a ROM pack. (ROM stands for Read Only Memory.) ROM packs are normally plugged into the Graphic System via slots located on the rear panel of the main chassis.

Typically, the BASIC interpreter "sees" ROM pack instructions as special functions to be executed. For example, if a ROM pack contains a routine which squares a number, then the statement — CALL "SQUARE",A — causes the system to follow the instructions in the "SQUARE" routine; the number assigned to the variable A is squared and the result is reassigned to A. In each case, the name of the special routine, the parameter required in the CALL statement, and the function the routine performs is predefined.

In some cases, the characteristics of the system actually change while a special routine is being executed. For example, when an optional data communications interface routine is executed, the system's ability to execute BASIC is momentarily inhibited while the system adapts its operating characteristic to the data communications application at hand. After the operation is complete, the system's ability to execute BASIC is restored.

Making a Paper Copy of Displayed Information

The COPY statement is a direct command to an attached Hard Copy Unit to make a paper copy of information stored on the GS display. The COPY statement performs the same function as the MAKE COPY key on the GS keyboard.

Returning the Alphanumeric Cursor to the Home Position

The HOME statement causes the alphanumeric cursor (or the writing tool of an external peripheral device) to return to the home position. The HOME statement performs the same function as the HOME key on the GS keyboard.

Erasing the Screen and Returning the Cursor to the Home Position

The PAGE statement erases the GS display screen and returns the alphanumeric cursor to the home position (near the upper left-hand corner). The PAGE statement performs the same function as the PAGE key on the GS keyboard.

THE CALL STATEMENT

Syntax Form:

[Line number] CAL { string variable / string constant } [{ ; / , } { string constant / string variable / numeric expression }] ...

Descriptive Form:

[Line number] CALL routine name [{ ; / , } data item to be passed to firmware routine] ...

Purpose

The CALL statement transfers system control to the specified firmware routine.

Explanation

Firmware Routine Defined

A firmware routine is a special set of instructions which give the Graphic System the ability to execute special functions. Normally, one or more firmware routines are stored in a memory chip and packaged in a plastic housing called a ROM (Read Only Memory) Pack. One ROM pack holds a maximum of 16K bytes of instructions (1K=1Ø24 bytes).

Specifying a Routine Name

Each firmware routine in a ROM pack has a preassigned name of six characters or less. Routine names are contained in a directory which is permanently fixed in the ROM pack when it is manufactured. The name of a routine can be specified as a string constant in the CALL statement or assigned to a string variable and specified as a string variable. For example:

```
2ØØ CALL "OOOOPS"
21Ø A$ = "OOOOPS"
22Ø CALL A$
```

In line 200, the routine name OOOOPS is specified as a string constant. Notice that the routine name is enclosed in quotation marks when specified as a string constant. In lines 210 and 220, the same routine is called again only this time the name is assigned to a string variable in line 210 and specified as a string variable in the CALL statement (line 220).

If the routine name is specified as more than six characters, the BASIC interpreter ignores the additional characters. For example:

```
230 CALL "OOOOPS-A-DAISY"
```

When this statement is executed, the same routine "OOOOPS" is called; in this case, however, the BASIC interpreter ignores the additional characters "-A-DAISY."

If the predefined routine name has less than six characters, then only those characters in the name can be specified; trailing blanks may be omitted.

Transfering System Control

The CALL statement is used to transfer system control to a specialized firmware routine as follows:

```
200 CALL "EDITOR"
```

When this statement is executed, the BASIC interpreter searches through the directory of each ROM pack for a routine called "EDITOR." When the routine "EDITOR" is found, system control is passed to that routine. When "EDITOR" is finished executing, system control is passed back to the BASIC interpreter and the next statement in the BASIC program is executed. If "EDITOR" is not found, an error occurs and program execution is aborted.

Passing Data Items to the Firmware Routine

If data items are specified in the CALL statement, then those data items are passed to the firmware routine as the routine is executing. For example:

```
250 CALL "FIX IT",295.5,Z$,M6
```

When this statement is executed, the numeric constant 295.5 is passed first to the routine "FIX IT" as it is executing. The character string assigned to Z$ is passed next, then the numeric value assigned to M6. The meaning of these data items is dependent on the definition and purpose of the routine. Complete instructions for using a routine are provided with each ROM pack.

THE COPY STATEMENT

Syntax Form:

[Line number] COP

Descriptive Form:

[Line number] COPY

Purpose

The COPY statement causes an attached Hard Copy Unit to make a paper copy of information on the GS display.

Explanation

The COPY statement performs the same function as the MAKE COPY key on the GS keyboard. When the COPY statement is executed, a MAKE COPY control signal is sent to an option Hard Copy Unit (if attached). For example:

```
500 COPY
```

When this statement is executed under program control, program execution stops while an attached Hard Copy Unit makes a scan of the GS display screen. The information on the screen is then reproduced on paper and presented to the Graphic System operator. Processing continues as soon as the scan is completed.

THE HOME STATEMENT

Syntax Form:

[Line number] HOM [I/O address]

Descriptive Form:

[Line number] HOME [I/O address]

Purpose

The HOME statement returns the alphanumeric cursor or the writing tool of an external peripheral device to the HOME position. The HOME position on the GS display is near the upper left-hand corner.

Explanation

The GS DISPLAY

The HOME statement performs the same function as the HOME key on the GS keyboard. For example:

```
150 HOME
```

When this statement is executed under program control, the alphanumeric cursor returns to the HOME position. The display is not erased.

External Peripheral Devices

If an I/O address is specified in a HOME statement, then a HOME command is set to the specified peripheral device over the General Purpose Interface Bus (GPIB). For example:

```
160 HOME @4:
```

When this statment is executed, the I/O address @4,23: is issued over the GPIB. Primary address 4 tells peripheral device number 4 that it has been selected to take part in the upcoming I/O operation. Secondary address 23 is issued by default and tells peripheral number 4 to executed a HOME command. In this case, data is not transfered after the I/O address is issued, so the I/O operation is terminated by sending the universal commands UNTALK and UNLISTEN over the GPIB.

CTRL ↑ Homes the Alphanumeric Cursor on the GS Display

Sending the ASCII control character "CTRL ↑" to the GS display also executes a HOME command. For example:

```
170 PRINT "↑4051 GRAPHIC SYSTEM"
```

When this statement is executed, the CTRL ↑ character (Record Separator) Homes the alphanumeric cursor. The character string "4051 GRAPHIC SYSTEM" is then printed on the screen starting at the HOME position. The control ↑ character is entered from the GS keyboard by pressing the CTRL key and at the same time pressing the ↑ key. For more information on sending control characters to the GS display, refer to the PRINT statement in the Input/Output Operations Section of this manual.

THE PAGE STATEMENT

Syntax Form:

[Line number] PAG [I/O address]

Descriptive Form:

[Line number] PAGE [I/O address]

Purpose

The PAGE statement erases the GS display and returns the alphanumeric cursor to the HOME position.

Explanation

The GS Display

The PAGE statement performs the same function as pressing the PAGE key on the GS keyboard; the screen is erased and the alphanumeric cursor returns to the HOME position. For example:

```
260 PAGE
```

When this statement is executed under program control, the GS display flashes as the screen is cleared. After the flash, the alphanumeric cursor appears in the upper left-hand corner.

External Peripheral Devices

If an I/O address is specified in a PAGE statement, then a PAGE command is sent to the specified peripheral device over the General Purpose Interface Bus (GPIB). For example:

```
270 PAGE @15:
```

When this statement is executed under program control, the I/O address @15,22: is issued over the GPIB. Primary address 15 tells peripheral device number 15 that it has been selected to take part in the upcoming I/O operation. Secondary address 22 is issued by default and tells device

15 to execute a PAGE command. In this case, data is not transfered after the I/O address is issued, so the I/O operation is terminated by sending the universal commands UNTALK and UNLISTEN over the GPIB.

CTRL L Pages the GS Display

Sending the ASCII control character "CTRL L" to the GS display also executes a PAGE command. For example:

```
28Ø PRINT "L4051 GRAPHIC SYSTEM"
```

When this statement is executed, the CTRL L character (Form Feed) pages the GS display. The character string "4051 GRAPHIC SYSTEM" is then printed on the screen. The control L character is entered from the keyboard by pressing the CTRL key and at the same time pressing the L key. (For more information on sending control characters to the GS display, refer to the PRINT statement in the Input/Output Operations Section of this manual.)

MEMORY MANAGEMENT

Section 4

MEMORY MANAGEMENT

INTRODUCTION TO MEMORY MANAGEMENT

The memory management functions in this section enable you to keep track of the memory bytes used to store the current BASIC program and the number of free memory bytes remaining in the system.

Deleting Items Stored in Memory

The DELETE statement gives you freedom to remove BASIC statements and variables stored in the read/write random access memory. Both variables and program lines can be deleted.

How Much Free Memory is Left?

The MEMORY function returns the number of free memory bytes remaining. As an added feature, this function performs a memory compress before it returns the number of free bytes. During the memory compress, all the fragmented portions of memory are reclaimed and made available for re-assignment. Because MEMORY is a numeric function returning a numeric result, it can be specified in a numeric expression.

How Much Memory Space Does the Program Take Up?

The SPACE function returns the maximum number of bytes of memory required to store the current BASIC program. This information must be obtained before the program is stored on the external medium such as a magnetic tape. Because SPACE is a numeric function returning a numeric result, it can be specified in a numeric expression.

The amount of memory required to run a program is quite different from the amount required to store a program. No space is required for variables, arrays, or strings until they are assigned or dimensioned. Also, accessory ROM packs or option interfaces require some memory space for storing data and environmental parameters.

THE DELETE STATEMENT

Syntax Form:

[Line number] DEL { ALL / variable list / line number [, line number] }

Descriptive Form:

[Line number] DELETE { ALL (entire memory) / variables to be deleted / line number [starting , line number ending] }

Purpose

The DELETE statement logically removes the specified BASIC statements or the specified variables from the read/write random access memory. If DELETE ALL is specified, then the entire random access memory is cleared.

Explanation

Deleting Variables

If the statement DELETE ALL is executed, the BASIC interpreter clears the entire program, including defined variables, from the read/write random access memory and executes an END statement. An INIT command is also executed.

If variables are specified as parameters in the DELETE statement, such as...

```
DELETE A, B, C$, D5, E1
```

then the assigned values of the specified variables are cleared, and the variables enter an undefined state.

Deleting Program Statements

If a line number is specified as the parameter in a DELETE statement, such as...

```
DELETE 5ØØ
```

then the specified statement is cleared from memory; in this example, statement 5ØØ is deleted and cannot be recovered.

If two line numbers are specified as parameters in a DELETE statement, such as the statement...

```
DELETE 500, 1000
```

then all the statements between the specified statements are logically removed from memory; in this case, statements 500 through 1000 are removed from memory and cannot be recovered. In addition, the following statement is allowed:

```
500 DELETE 400, 600
```

In this case, all statements from 400 through 600 (including the DELETE statement) are cleared from memory under program control.

A Word of Caution

Once a DELETE statement is executed, the deleted information can not be recovered unless it is first stored on an external media such as magnetic tape. Refer to the Input/Output Operations section for information on storing programs and data on an external media.

NOTE

The deleted information is not actually removed from memory until the system needs additional memory space. However, the deleted information is "tagged" as such and cannot be recovered. This is what is meant by the phrase "logically removed from memory."

THE MEMORY FUNCTION

Syntax Form:

MEM

Purpose

The MEMORY function forces the BASIC interpreter to combine the available free memory into one contiguous block and return the number of free bytes remaining.

Explanation

Immediate Execution

To find out how much free memory remains in the read/write random access memory, type in the following statement and press the RETURN key on the GS keyboard.

```
MEM
```

Program Execution

Because the MEMORY function is a numeric function, it can be specified as part of a numeric expression and evaluated under program control. For example:

```
410 LET M = MEM/8
```

When this statement is executed, the number representing the number of free memory bytes is divided by 8 and assigned to the variable M.

When the system is first turned on and the memory is empty, MEM + 2000 is approximately the maximum memory capacity. The first 2K bytes of memory are reserved by the microprocessor for a working area; the remaining bytes are used for storing BASIC program instructions and data.

Finding Out How Much Memory Space is Reserved for Data

Once a BASIC program is loaded into memory, the MEM function tells you how much space is left for storing data. During the course of program execution, variables may sometimes be

dimensioned larger than the available memory space and a fatal error occurs. The following guidelines are provided to help you figure out how much memory is taken up by each data item, so if you get a MEMORY FULL error condition, you can figure out where to cut corners to make it all fit. Here are the guidelines:

1. In general, any variable symbol entered into memory space takes up at least 13 bytes of memory space. If a variable is deleted from memory, 13 bytes of memory space are still reserved for the symbol name in a table listing. The only way to delete the symbol from the table and recover the 13 bytes is to execute a DELETE ALL statement, an OLD statement, or turn off the system power.

2. Numeric variable symbols plus their assigned scalar values take up 13 bytes of memory, regardless of the size of the scalar value. You don't save any memory space by deleting a numeric variable due to guideline number 1.

3. Each string variable entered into memory takes up the following memory space:

   String Dimension + 18 Bytes

   If the default dimension for a string variable (72) is used, then the variable takes up 90 memory bytes. If a string variable is dimensioned to a maximum of five characters, for example, the variable takes up 23 memory bytes, and so on.

4. Array variable take up the following memory space:

   Number of rows x Number of Columns x 8 + 18

THE SPACE FUNCTION

Syntax Form:

SPA

Purpose

The SPACE function returns the maximum number of bytes of storage required to store the current BASIC program in external ASCII format.

Explanation

To find out how much space is required to store the current program, type in the following statement and press the RETURN key:

```
SPACE
```

After SPACE is executed, the number of bytes required to store the current program is displayed. Once this information is known, a program file can be MARKed on a magnetic tape cartridge and the program stored on tape with the SAVE statement. (See MARK and SAVE in the Input/Output Operations section for more information.)

Because the SPACE function returns a numeric result, the function can be specified as part of a numeric expression. For example:

```
300 LET B = SPA + 50
```

When this statement is executed, the space required to store the current BASIC program is added to 50 and the result is assigned to the variable B.

The number returned by the SPACE function is only an approximation of the number of storage bytes required to store the current BASIC program. The BASIC interpreter arrives at this number by multiplying the number of program lines by 72 (the maximum number of characters per line plus one for the carriage return delimiter). If each line in the current program is 36 characters or less, for example, then the actual space required to store the program is less than one-half the numeric value returned by the SPACE function.

CONTROLLING PROGRAM FLOW

Section 5

CONTROLLING PROGRAM FLOW

INTRODUCTION

A BASIC program can contain any number of statements as long as the memory capacity of the system is not exceeded. Each statement is entered on a separate line with a line number. The BASIC interpreter keeps the statements in correct numeric sequence, even if they are not entered in sequence. The BASIC interpreter does not recognize a line number unless it is explicitly entered into memory with a valid BASIC statement. Normally, the statements are executed sequentially starting with the lowest line number in memory and proceeding to the highest line number in memory. This pattern can be altered, however, by exercising the statements described in this section.

Starting a Program

A BASIC program is started by executing the RUN statement directly from the GS keyboard. If a starting line number is not specified, the program automatically starts with the lowest number in memory.

Stopping a Program

Program execution is stopped by executing an END statement, a STOP statement, or a RETURN statement. The STOP and RETURN statements do not disturb the system environmental conditions. This allows the program to be continued from the stopping point by executing the RUN statement with the proper statement number following it. Pressing the BREAK key on the GS keyboard also stops program execution unless the internal magnetic tape unit is running.

Ending a Program

Program execution is ended and control is returned to the GS keyboard by executing an END statement. Once an END statement is executed, the program can't be continued; in most cases it must be restarted from the beginning. Pressing the BREAK key twice on the GS keyboard also terminates program execution.

Looping

Sometimes it is desirable to have a series of statements executed several times before continuing with sequential execution. The FOR and NEXT statements work together to control the number of times the BASIC interpreter executes a series of statements before proceeding with normal sequential execution.

Unconditional Branching

The GO TO statement unconditionally transfers program control to another point in the program. Once program control is transferred, normal sequential execution continues from that point.

Branching to a Subroutine

The GOSUB statement allows program control to be transferred from the main program to a program subroutine. After the subroutine is executed, the RETURN statement transfers program control back to the interruption point in the main program. Program subroutines are usually a series of statements which are executed frequently as the main program progresses through normal sequential execution.

Test and Branch

The IF . . . THEN . . . statement causes program control to be transferred to another point in the program, if a specified condition is logically true. The specified condition can be a relational comparison between two numeric expressions, a relational comparison between two character strings, or a logical comparison between two numeric expressions.

BREAK

Pressing the BREAK key on the GS keyboard causes program execution to terminate after the current instruction is finished executing. Pressing the BREAK key twice causes program execution to terminate immediately.

Interrupts

Interrupts are asynchronous events which cause GOSUB like actions to occur. Refer to the section on Handling Interrupts for complete information on internal and external interrupt conditions.

THE END STATEMENT

Syntax Form:

[Line number] END

Descriptive Form:

[Line number] END

Purpose

The END statement terminates program execution, closes all open files, and returns control to the GS keyboard.

Explanation

The END statement is normally the highest numbered line in a program, but doesn't have to be. When the END statement is executed by the BASIC interpreter, program execution is terminated with no printed indication; the line counter is reset to the lowest line number in memory, any open files are closed, and the internal execution stack is cleared. END statements may appear anywhere in a program. An END statement is automatically executed at the end of a program, if an END statement is not found.

THE FOR AND NEXT STATEMENTS

Syntax Form:

Line number FOR numeric variable = numeric expression TO numeric expression [STE numeric expression]

Descriptive Form:

Line number FOR index = starting value TO ending value [STEP increment for each loop]

Syntax Form:

Line number NEX numeric variable

Descriptive Form:

Line number NEXT index

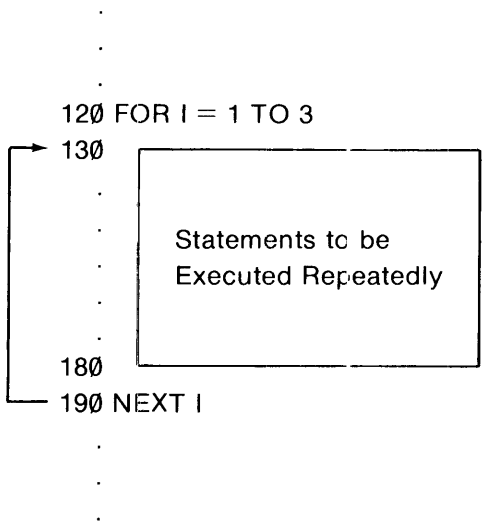
Purpose

The FOR and NEXT statements work together to control the number of times a section of a program is repeatedly executed. This technique is called looping.

Explanation

A Simple FOR/NEXT Loop

The following example illustrates a simple FOR/NEXT loop. The statements in the loop are executed three times before the program continues with normal sequential execution.



Line 12Ø in this example is the first statement in the loop. This statement controls the number of times the loop is executed. In line 12Ø, the numeric variable I is specified as the index and is assigned a starting value of I. The ending value of the I is specified as 3 and, because a STEP is not specified, a STEP of +1 is assumed by default.

After the FOR statement is evaluated in line 12Ø, the BASIC interpreter executes lines 13Ø through 18Ø in sequence. During this time the assigned value of I remains at 1, the initial starting value. When line 19Ø is executed, the BASIC interpreter adds +1 to the value of I and compares the new value (2) to the ending value specified in line 12Ø. In this case, the new value (2) is not greater than the specified ending value (3), so program control is transferred back to line 13Ø and lines 13Ø through 18Ø are re-executed. As these statements are executed, the assigned value of I remains at 2.

At the end of the second pass through the loop, line 19Ø is re-executed. Again, the increment +1 is added to the current value of I and compared to the ending value specified in line 12Ø. This time the new value (3) is equal to the specified ending value, but not greater than the specified ending value; therefore, program control is transferred back to line 13Ø and the statements in the loop are executed a third time. During this pass, the assigned value of I remains at 3.

At the end of the third pass, the increment +1 is added to the current value of I and the new value (4) is found to be greater than the specified ending value (3). Program control is then transferred to the statement which follows the NEXT I statement and the program continues executing in sequence.

NOTE

The final value of the index does not equal the ending value of the FOR/NEXT loop.

The index in this example is specified as the numeric variable I, however any valid numeric variable symbol can be used as long as the same symbol is used in both the FOR and the NEXT statements.

The starting value and ending value of the index can be specified as a numeric expression, if desired, as long as the variables in the numeric expressions (if any) have assigned values by the time the FOR statement is executed.

In addition, the number of statements in the loop is not restricted. Any number of statements can be included in the loop (within the limits of memory).

Specifying a STEP

The following example illustrates a simple FOR/NEXT loop with a specified STEP.

```
.
.
.
200 FOR I = 0 to 10 STEP 2
210
.
.
.
.
.
.
.
.
.
.
250
260 NEXT I
270 PRINT "WE HAVE JUST MADE AN EXIT FROM A LOOP"
.
.
.
```

Statements to be Executed Repeatedly

When line 200 in this example is executed, the numeric variable I is assigned a starting value of 0, the ending value of I is specified as 10, and the STEP is specified as 2. After line 200 is evaluated, lines 210 through 250 are executed in sequence. As these statements are executing, the assigned value of I remains at 0, the initial starting value.

NOTE

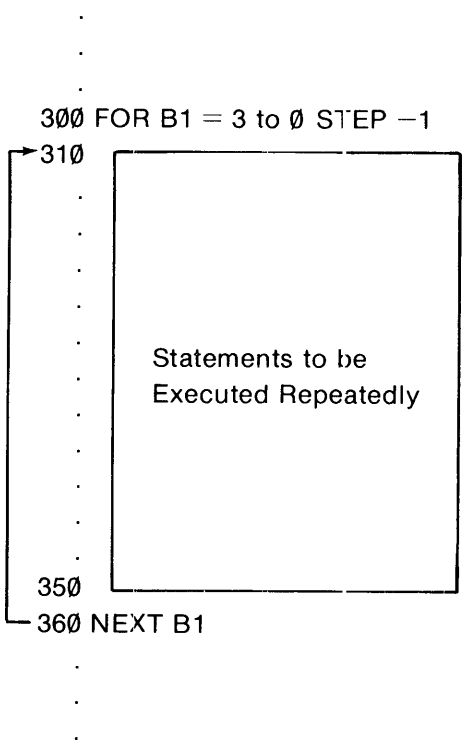
After the FOR statement is evaluated, at the beginning of the LOOP sequence, the index starting value, ending value, and step increment are placed in temporary storage and are not evaluated again.

When line 26Ø is executed, the specified STEP (2) is added to the assigned value of I and compared to the specified ending value (1Ø). The new value of I is not greater than the specified ending value (within the parameters of FUZZ), so program control is transferred back to the statement following the FOR statement (line 21Ø in this case) and lines 21Ø through 25Ø are re-executed.

Each time the NEXT I statement is encountered, the increment (2) is added to the current value of I and compared to the specified ending value 1Ø. The first time through the loop I=Ø; the second time through the loop I=2; the th rd time through the loop I=4; the fourth time through the loop I=6; the fifth time through the loop I=8; and the sixth time through the loop I=1Ø. At the end of the sixth pass, the increment 2 is added to 1Ø (the current value of I) and the result 12 is assigned to I. The new value of I (12) is compared to the specified ending value (1Ø) and found to be greater (within the limits specified by the FUZZ statement). Program control is then transferred to the statement following the NEXT statement (line 27Ø) and the program continues executing in sequence.

Using a Countdown Technique

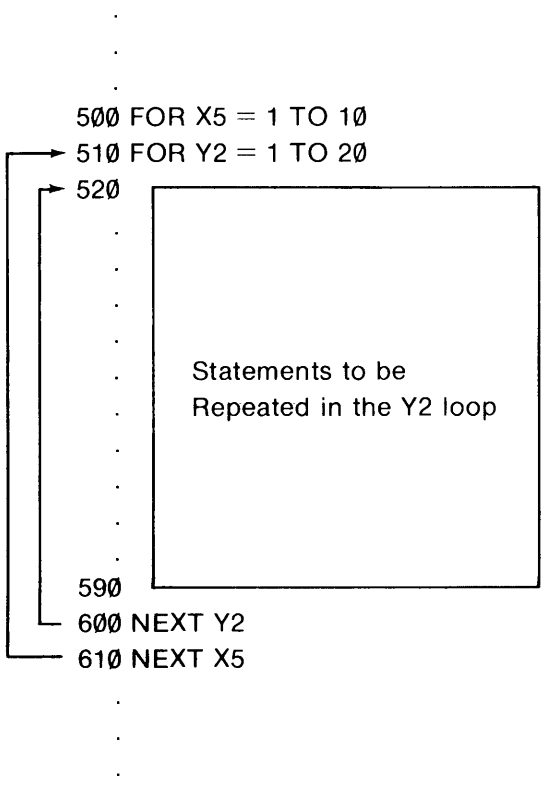
The STEP in the FOR statement can be specified as a negative number as well as a positive number as the following example illustrates:



In this example, the numeric variable B1 is used as the index and is assigned a starting value of 3 in line 300. The ending value of B1 is specified as 0 and the STEP is specified as −1. After line 300 is evaluated, lines 310 through 350 are executed in sequential order. During this time, the assigned value of B1 remains at 3 (the starting value). When NEXT B1 is executed in line 360, the step (−1) is added to the current value of B1 and compared to the ending value (0). If the new value of B1 is **less** than the specified ending value, then program control is transferred to the statement following the NEXT B1 statement; in this case it is not, so program control is transferred back to line 310 and lines 310 through 350 are re-executed. During the first pass, B1 has an assigned value of 3 (the starting value); during the second pass B1 has an assigned value of 2; during the third pass, B1 has an assigned value of 1; and during the fourth pass B1 has an assigned value of 0. At the end of the fourth pass, −1 is added to 0 (the current value of B1) and the result (−1) is compared to the specified ending value (0). Minus 1 is found to be less than 0, so program control is transferred to the statement following line 360 and the program continues executing in sequence.

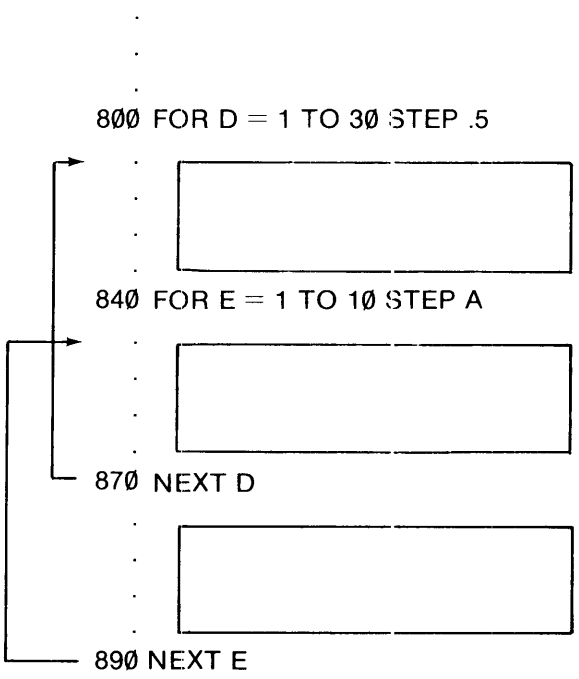
Nesting FOR/NEXT Loops

FOR/NEXT loops can be nested inside each other as shown below:



In this example, lines 52Ø through 59Ø in the Y2 loop are executed 2Ø times for each pass through the X5 loop. After the program makes an exit from the X5 loop, the statements in the Y2 loop will have been executed 2ØØ times.

FOR/NEXT loops cannot "cross." The following is an illegal operation:



Branching Into and Out of a FOR/NEXT Loop

Branching out of a FOR/NEXT loop using the GOTO,GOSUB,or IF...THEN...statement and not returning to the exit point is a legal practice, and an error doesn't occur, but information pertaining to the exit point accumulates in memory and isn't cleared until an END statement is executed. Branching randomly into a FOR/NEXT loop from another point in the program is dangerous programming practice. If a NEXT statement is executed without first executing a FOR statement, then an error occurs and program execution is aborted.

Branching out of a FOR/NEXT loop is a legal practice; however, it is not recommended. The execution of a FOR statement dynamically allocates 26 bytes of memory to store loop information. When the FOR/NEXT loop is satisfied (i.e., performs a normal exit) the 26 bytes of dynamically allocated memory is freed.

Branching out of a FOR/NEXT loop with an IF-THEN or a GOTO statement prevents a normal exit and the 26 bytes of memory is not freed for future use.

THE GOSUB AND RETURN STATEMENTS

Syntax Form:

[Line number] GOS { line number / numeric expression OF line number [, line number] ... }

Descriptive Form:

[Line number] GOSUB { line number / line number selector OF line number list }

Syntax Form:

[Line number] RET

Descriptive Form:

[Line number] RETURN

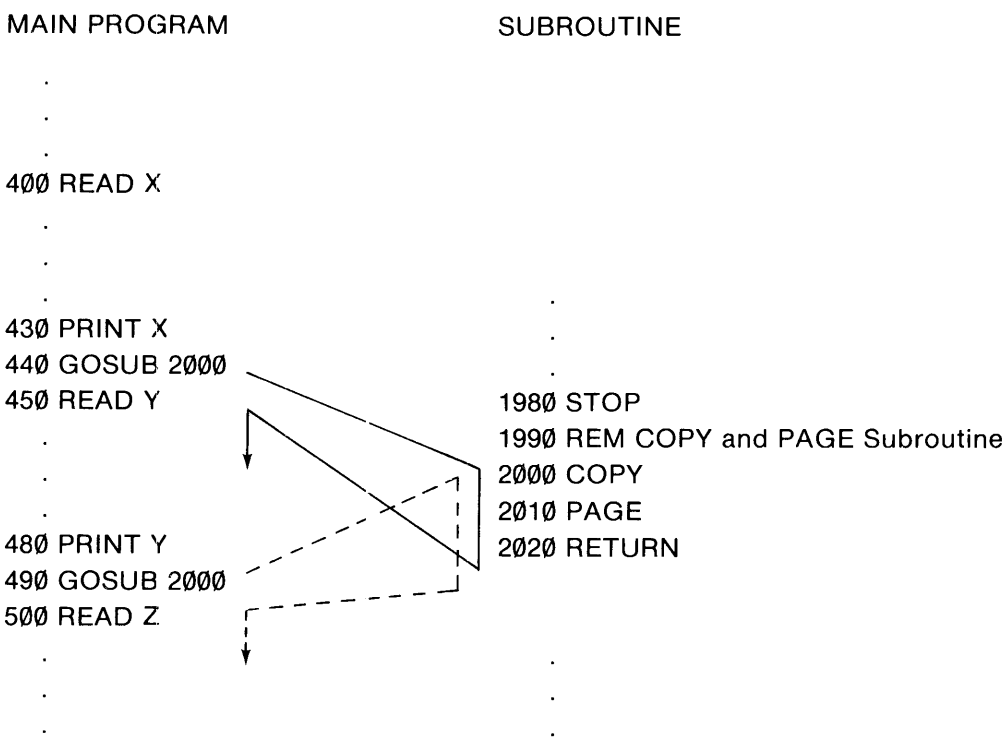
Purpose

The GOSUB statement transfers program control to the beginning statement of a program subroutine. The RETURN statement then transfers program control back to the statement which follows the GOSUB statement. If the GOSUB . . . OF . . . form of the GOSUB statement is used, then program control is transferred to the beginning statement of a subroutine indicated by a line number selector. The RETURN statement at the end of the subroutine then transfers program control back to the statement which follows the GOSUB . . . OF . . . statement.

Explanation

The GOSUB statement allows a frequently used set of program instructions to be entered as a program subroutine, then allows these instructions to be executed at different points in the main program whenever the need arises.

The following example illustrates program flow when a GOSUB statement is executed:



In this example, line 400 in the main program causes the BASIC interpreter to assign values to the elements of the previously dimensioned array variable X. These values are contained in a DATA statement located elsewhere in the program. A short time later, line 430 is executed and the elements of array X are printed on the GS display. After the elements are printed, line 440 transfers program control to line 2000 — the beginning of a PAGE and COPY subroutine. This subroutine causes an attached Hard Copy Unit to make a paper copy of the information on the display (line 2000), then PAGE the cursor to the HOME position and erase the screen. The RETURN statement in line 2020 transfers program control back to the main program — to the statement following the GOSUB statement which originally transferred control. (The RETURN statement is not to be confused with the RETURN key on the GS KEYBOARD.)

It can be seen from this example that the COPY and PAGE subroutine can be executed anywhere in the program just by specifying a GOSUB 2000 statement.

Selecting a Subroutine from a List of Subroutines

The GOSUB . . . OF . . form of the GOSUB statement provides the option of transferring program control to one of several subroutines listed in the GOSUB statement. The subroutine selected depends on the line number selector which follows the keyword GOSUB.

For example:

```
300 GOSUB D OF 500, 600, 700, 800
```

In this example, the assigned value of the numeric variable D is the line number selector which indicates which subroutine in the line number list following the keyword OF is to be executed. Each line number in the list is the first line number in a subroutine. If the assigned value of D is 1, for example, then program control is transferred to the subroutine which starts with the line number 500. If the assigned value of D is 2, then program control is transferred to the subroutine which starts with the line number 600, and so on. After the RETURN statement is executed at the end of the subroutine, program control is transferred back to the statement which immediately follows the GOSUB . . . OF . . . statement which originally transferred control.

If the assigned value of D in the preceding example is not an integer (3.333 for example) then the BASIC interpreter rounds the value to an integer before a selection is made. Any number with a fractional part equal to or greater than .5 is rounded to the next highest integer. (This rounding is done for selection purposes only; the actual value of D remains unchanged.)

If the line number selector rounds to an integer which is less than 1 or greater than the number of line numbers in the list, then the branch does not occur and program control is transferred to the next statement following the GOSUB . . . OF . . . statement.

Internal Considerations

Each time a GOSUB statement is executed, 6 bytes of memory are dynamically allocated to store the return address. This information is used by the RETURN to branch back to main program and the 6 bytes of memory are then freed for future use.

Use of GOTO and IF-THEN statements which branch to line numbers within the scope of a subroutine is legal and acceptable practice. It is also acceptable to have several RETURN statements in a subroutine when required by the application's logic.

However, branching out of a subroutine back to the main program or another part of the main program with a GOTO or IF-THEN statement is not recommended. This practice will eventually generate a MEMORY FULL condition since the six bytes of memory are not freed by the execution of a RETURN statement.

The execution of an END statement will also free all the dynamically allocated memory.

THE GO TO STATEMENT

Syntax Form:

[Line number] { GO TO / GOTO } { line number / numeric expression OF line number [, line number] ... }

Descriptive Form:

[Line number] { GO TO / GOTO } { line number / line number selector OF line number list }

Purpose

The GO TO statement unconditionally transfers program control to the specified line number. If the keyword OF is specified, then program control is transferred to a line number located in a line number list.

Explanation

Transferring Program Control

The GO TO statement unconditionally transfers program control to the specified line number. After program control is transferred, normal sequential execution continues from that point. If the specified line number doesn't exist, an error occurs and program execution is aborted. The keyword GO TO can also be specified as GOTO.

The following example is a program which solves the Pythagorean Theorem after the values of A and B are input from the GS keyboard. The GO TO statement in line 16Ø places the program into a continuous loop. This allows the theorem to be solved again and again without restarting the program each time. The only way to exit this loop is to press the BREAK key on the GS keyboard.

```
1ØØ PRINT "Let's Solve the Pythagorean Theorem"
11Ø PRINT  "C = SQR (A↑2 + B↑2)
12Ø PRINT "Enter a Value for A and B"
13Ø INPUT A,B
14Ø PRINT "C is Equal to "; SQR (A↑2 + B↑2)
15Ø PRINT
16Ø GO TO 13Ø
```

Selecting the Transfer Destination Point from a List

The GO TO . . . OF . . . form of the GO TO statement allows the transfer destination point to be selected from a list of line numbers. For example, the statement . . .

```
25Ø GO TO 2 OF 9ØØ, 545, 25, 365
```

causes program control to be transferred to line 545, the second statement in the list. If this statement were . . .

```
25Ø GO TO 4  OF 9ØØ, 545, 25, 365
```

then program control would be transferred to line 365, the fourth statement in the list.

The line number selector which follows the keyword GO TO is usually specified as a numeric expression (the selector can also be numeric variable, a numeric function, a subscripted array variable, etc.) The only requirement is that all variables in the numeric expression must have assigned values by the time the statement is executed.

If the line number selector is specified as a variable, and the assigned value of the variable is not an integer, then the BASIC interpreter rounds the value to an integer before selecting the line number. For example:

```
375 GO TO R3 OF 1Ø5, 215, 53Ø, 135
```

Assume the value of R3 in this statement is dependent on the result of a previously executed mathematical equation. If the assigned value of R3 turns out to be 3.67, for example, the BASIC interpreter transfers program control to statement 135, the fourth statement in the list. Any number with a fractional part equal to or greater than .5 is rounded to the next highest integer. The rounding of the line number selector is done for selection purposes only; this process does not alter the original value assigned to the variable.

If the line number selector rounds to an integer which is less than 1 or greater than the number of line numbers in the list, the branch does not occur and program control is transferred to the statement which follows the GO TO statement.

GO TO should not be used to enter FOR/NEXT loops; doing so may produce unpredictable results or fatal errors.

A good example on how the GO TO . . . OF . . . statement is used is found under the POLL statement explanation in the Handling Interrupts section.

Using the GO TO Statement and the STEP PROGRAM Key

When a GO TO statement is entered directly from the GS keyboard and the RETURN key is pressed, the program line counter is set to the specified line number. For example:

```
GO TO 5ØØ
```

This statement sets the program line counter to line 5ØØ. Line 5ØØ can now be executed by pressing the STEP PROGRAM key on the GS keyboard. Pressing the STEP PROGRAM key repeatedly causes the program to be executed sequentially one statement at a time from the point where the program line counter is set. This technique is useful in trying to find a bug in a program. For example, the GO TO statement can be used to place the program line counter in the area of the program to be debugged. The execution of the program can then be examined one statement at a time as the STEP PROGRAM key is pressed again and again. See the Program Editing section for more information on debugging a program.

THE IF ... THEN ... STATEMENT

Syntax Form:

[Line number] IF numeric expression THE line number

Descriptive Form:

[Line number] IF numeric expression THEN line number

Purpose

The IF . . . THEN . . . statement transfers program control to the specified line number if the specified numeric expression is logically true; if the numeric expression is not logically true, then the program continues executing in sequence.

Explanation

The IF . . . THEN statement is a conditional transfer statement which "tests" to see if a specified numeric expression is true. Typically, the numeric expression is a comparison between two numeric expressions or a comparison between two character strings. IF the specified numeric expression is logically true, THEN the program branches to the specified line number and continues executing in sequence. IF the specified numeric expression is not logically true, THEN the branch does not occur and the program continues to the next statement.

A Simple Numeric Expression

The simplest form of the IF . . . THEN . . . statement involves a numeric expression without logical or relational operators. For example:

```
300 IF A THEN 500
```

In this example, the assigned value of A is evaluated to see if it is a logical one or a logical zero. IF the absolute value of A is equal to or greater than .5, then it is considered to be a logical one and program control is transferred to line number 500. IF the absolute value of A is less than .5, then it is considered to be a logical zero and program control continues to the statement following the IF . . . THEN . . . statement.

The numeric expression following the keyword IF can contain any number of numeric constants, numeric variables, numeric functions, and subscripted array variables joined together by arithmetic, logical, and relational operators. The only requirement is that the numeric expression must be in a form such that the BASIC interpreter can reduce the expression to a numeric constant. The BASIC interpreter treats the numeric constant as a logical one or a logical zero.

Comparing the Relationship Between Two Numeric Expressions

The numeric expression following the keyword IF can contain two numeric expressions separated by a relational operator. This type of numeric expression takes the form:

numeric expression relational operator numeric expression

Of course, the numeric expression on either side of the relational operator can be a combination of numeric functions, numeric variables, subscripted array variables, and numeric constants. All numeric comparisons are made within the parameters of FUZZ. The relational operator can be one of the following:

RELATIONAL OPERATOR	MEANING
<>	"is not equal to"
<	"is less than"
<=	"is less than or equal to"
>	"is greater than"
>=	"is greater than or equal to"

The following examples illustrate how the BASIC interpreter interprets a conditional transfer statement:

STATEMENT	MEANING
45ØØ IF M=R THEN 6ØØ	IF the current value of M is equal to the current value of R, THEN go to line 6ØØ; if not, proceed to the next statement.
455 IF M<>R THEN 7ØØ	IF the current value of M is not equal to the current value of R, THEN go to line 7ØØ; if it is, proceed to the next statement.
46Ø IF X<MEM THEN 8ØØ	IF the current value of X is less than the current value of the MEMORY function, THEN go to line 8ØØ; if not, proceed to the next statement.

STATEMENT	MEANING
465 IF D3< = X↑2+3 THEN 9ØØ	IF the current value of D3 is less than or equal to the current value of the expression X↑2+3, THEN go to line 9ØØ; if not, proceed to the next statement.
47Ø IF Q9−45>Ø THEN 1ØØØ	IF the current value of the expression Q9−45 is greater than Ø, THEN go to line 1ØØØ; if not, proceed to the next statement.

Logical Comparisons

Logical comparisons can also be used as a basis for conditional transfers. The logical comparisons are specified as follows:

numeric expression logical operator numeric expression

Any number of numeric expressions and logical operators can be specified. The rules of Boolean algebra must be followed. Each numeric expression is treated as a logical one or a logical zero. All numbers whose absolute value is equal to or greater than .5 are considered to be a logical one; all numbers whose absolute value is less than .5 are considered to be a logical zero.

The logical operators are specified as AND, OR, or NOT and correspond to their Boolean algebra equivalent. The following examples illustrate conditional transfers based on logical comparisons:

STATEMENT	MEANING
25Ø IF A AND B THEN 6ØØ	IF the assigned value of A is a logical one AND the assigned value of B is a logical one, THEN go to line 6ØØ; if not, proceed to the next statement.
26Ø IF B OR C THEN 7ØØ	IF the assigned value of B is a logical one OR the assigned value of C is a logical one, THEN go to line 7ØØ; if not, proceed to the next statement.

STATEMENT	MEANING
27Ø IF A AND B OR NOT C THEN 8ØØ	IF the assigned value of A is a logical one AND the assigned value of B is a logical one OR the assigned value of C is NOT a logical one, THEN go to line 8ØØ; if not, then proceed to the next statement.

Execution Priority

The following list specifies the execution priority the BASIC interpreter follows when executing a BASIC statement. The highest priority is 1, the lowest priority is 14.

Priority	Operators
1.	Left Paren (
2.	Functions
3.	Monadic Operators +, −, and NOT
4.	Exponentiations Operators ↑
5.	Dyadic Operators * and /
6.	Dyadic Operators + and −
7.	The Arithmetic Operators MIN and MAX
8.	Relational Operators =, < >, <,>,< =, and > =
9.	The Logical Operators AND and OR
1Ø.	The Keyword USING and comma (,)
11.	Right Paren) and semicolon (;)
12.	The Keywords OF, THEN, STEP, TO, and the symbols @ # % =
13.	All Other Keywords
14.	Carriage Return

Comparing String Constants

The relationship between two string constants can also be used as the basis for a conditional transfer. A string comparison is allowed because the entry, as a whole, is reduced to a logical one or a logical zero. A string relationship is specified as follows:

{string constant / string variable} relational operator {string constant / string variable}

The string constant on either side or the relational operator can be specified as a string constant in quotation marks or represented by a string variable. The relational operators are the same as those used with numeric expressions.

The two string constants are compared one character at a time from left to right and evaluated according to the priority established in the ASCII Character Priority for String Inequalities Chart in Appendix B; the first difference determines the relationship. For example:

"Bugs" = "Bunny"

When this expression is evaluated, the BASIC interpreter first compares B and B and finds no difference. The second character in each string is then compared (u and u) and again no difference is found. The third characters are compared (g and n) and a difference is found. Since the letter n is considered greater (higher priority) than the letter g, the string constant "Bunny" is considered greater than the string constant "Bugs". The relationship as stated is therefore not true. If the comparison were . . .

"Bugs" < "Bunny"

then the relationship would be true.

When two string constants are compared, upper and lower case alphabetics are considered equal unless the statement SET NOCASE is executed. For example, madness = MADNESS unless NOCASE is set.

If two string constants are identical in every way except that one is longer than the other, then the longer string is considered the greater string.

As an example, the comparison "Rabbits" > "Rabbit" is true, because the string constant "Rabbits" has an additional letter (s).

The following examples illustrate ways in which string relationships can be used to control conditional transfers:

STATEMENT	MEANING
800 IF A$ = "Record" THEN 1200	IF the string constant assigned to A$ is equal to the string constant "Record", THEN go to line 1200; if not, proceed to the next statement.
810 IF A$ <> "Menu" THEN 1400	IF the string constant assigned to A$ is not equal to the string constant "Menu", THEN go to line 1400; if not, proceed to the next statement.

STATEMENT	MEANING
82Ø IF X$<Y$ THEN 3ØØØ	IF the string constant assigned to X$ is less than the string constant assigned to Y$, THEN go to line 3ØØØ; if not, proceed to the next statement.
83Ø IF "Rick" < = Q$ THEN 1ØØ	IF the string constant "Rick" is less than or equal to the string constant assigned to Q$, THEN go to line 1ØØ; if not, proceed to the next statement.
84Ø IF Z$> "Payroll" THEN 5ØØ5	IF the string constant assigned to Z$ is greater than the string constant "Payroll", THEN go to line 5ØØ5; if not, proceed to the next statement.
85Ø IF RS> =E$ THEN 999	IF the string constant assigned to R$ is greater than or equal to the string constant assigned E$, THEN go to line 999; if not, proceed to the next statement.

THE RETURN STATEMENT

Syntax Form:

[Line number] RET

Descriptive Form:

[Line number] RETURN

Purpose

If used alone, the RETURN statement returns the system to keyboard control (see description under the GOSUB statement).

Explanation

If a RETURN statement is encountered in a program and the BASIC interpreter is not currently executing a subroutine statement (GOSUB or GOSUB . . . OF), then program execution is terminated and control is returned to the GS keyboard. This is the same as executing an END statement.

THE RUN STATEMENT

Syntax Form:

[Line number] RUN [line number]

Descriptive Form:

[Line number] RUN [starting line number]

Purpose

The RUN statement places the system under program control.

Explanation

When RUN is executed directly from the GS keyboard, the BASIC interpreter executes a RESTORE, sets NOKEY, and clears all return vectors for GOSUB statements and FOR . . . NEXT loops. The BASIC interpreter then enters the program RUN mode and executes the lowest numbered statement as the first instruction.

If a line number is specified as a parameter, such as RUN 5ØØ, then the BASIC interpreter starts executing the program at the specified line number; in this case, the BASIC interpreter starts executing the program at line number 5ØØ. Variables defined in the statements which are skipped are considered to be undefined or retain their previous values, if values have been assigned to them.

The RUN statement in Graphic System BASIC language is different from the RUN command in most other BASIC languages in that it can be issued under program control. For example:

```
55Ø RUN
```

When this statement is executed, most system parameters are set to their default values (as specified in the INIT statement) and program control is transferred to the lowest line number in memory.

If the statement . . .

```
56Ø RUN 3ØØ
```

is executed under program control, then the system parameters stay in their present state and program control is transferred to line 3ØØ. This statement is the same as executing a GOTO 3ØØ statement.

A running program can be stopped by pressing the BREAK key on the GS keyboard. After the current statement is finished executing, the message

```
PROGRAM INTERRUPTED PRIOR TO LINE . . .
```

or

```
PROGRAM INTERRUPTED
```

is printed on the GS display and control is returned to the GS keyboard. The program can be continued from the interruption point by executing a RUN statement. For example, if the BREAK key is pressed and the message . . .

```
PROGRAM INTERRUPTED PRIOR TO LINE 35Ø
```

is printed on the screen, then a RUN 35Ø statement can be executed from the GS keyboard to continue the program at line 35Ø. The program environmental conditions remain unchanged.

If the BREAK key is pressed twice in succession before the current statement is finished executing, the message

```
PROGRAM ABORTED IN LINE . . .
```

is printed on the screen and program execution is ended. (Unless the internal magnetic tape unit is running). This is the same as executing an END statement.

THE STOP STATEMENT

Syntax Form:

[Line number] STO

Descriptive Form:

[Line number] STOP

Purpose

The STOP statement halts program execution and returns control of the system to the GS keyboard.

Explanation

If a STOP statement is executed under program control, the BASIC interpreter halts program execution and returns control to the GS keyboard. For example:

```
430 STOP
```

When this statement is executed, the program halts at line 430 and the message

```
STOP EXECUTED IN LINE 430 PRIOR TO LINE 440
```

is printed on the screen. This message indicates the present position of the line counter and the position of the next statement in the program. If there are no more statements in the program, the message "STOP EXECUTED IN LINE 430" is printed on the screen.

The STOP statement does not disturb the current values assigned to variables or the current environmental conditions; therefore in this example, the program can be continued by entering a RUN 440 statement directly from the GS keyboard and pressing the RETURN key.

HANDLING INTERRUPTS

Section 6

HANDLING INTERRUPTS

INTRODUCTION TO HANDLING INTERRUPTS

General

The current BASIC program must be designed to handle peripheral service requests and SIZE error conditions if they occur. This means that service routines must be included in the program to service the needs of each peripheral on the General Purpose Interface Bus; in addition, the executing program must be responsive to error conditions when they occur, or program execution is terminated immediately. The statements in this section provide an interrupt handling facility which enables program control to be transferred to a specified line number when an interrupt condition occurs, and then returned to the interruption point in the main program after the interrupt condition is serviced.

Interrupt Conditions

There are four interrupt conditions which a BASIC program can respond to. These conditions are listed under the heading Interrupt Conditions following this introduction.

When an Interrupt Occurs

The action taken when an interrupt condition occurs is specified in the ON...THEN... statement. An ON...THEN... statement must be executed for each interrupt condition.

When an ON...THEN... statement is evaluated during program execution, no immediate action may occur at that time; however, the ON...THEN... statement arms the system to respond to the specified interrupt condition. Program execution continues normally until the specified interrupt occurs; when it does, the BASIC interpreter finishes executing the current statement, then transfers program control to the ON...THEN statement. The ON...THEN... statement transfers control to a service subroutine. When the subroutine is finished executing, program control is transferred back to the interruption point in the main program — to the statement which would have been executed next if the interrupt hadn't occurred.

Polling External Peripheral Devices

External devices get the attention of the processor by pulling down on an SRQ (Service Request) signal line on the General Purpose Interface Bus. There is only one SRQ line on the General Purpose Interface Bus (GPIB), so each device must be polled to determine which device is requesting service.

Normally, the ON SRQ THEN... statement transfers program control to a POLL statement in the BASIC program. The POLL statement causes the BASIC interpreter to serially poll a list of devices on the GPIB. The order in which the devices are polled is specified in the POLL statement. Once the device which is requesting service is found, program control is transferred to a service routine for that device via a GOTO...OF... statement which generally follows the POLL statement.

When the service routine is finished executing, program control is returned to the interruption point in the main program.

Turning OFF a Program Response to an Interrupt Condition

In some phases of program execution, it is desirable to disable a program's response to an interrupt condition. This is done by executing an OFF statement for that interrupt condition. A program's response to the interrupt condition is re-enabled by re-executing an ON...THEN... statement.

Waiting for an Interrupt

Sometimes it is desirable to delay program execution and WAIT for an interrupt such as SRQ. The WAIT statement causes a program to wait for an interrupt (any interrupt). When an interrupt does occur, program execution resumes. Program control is then transferred immediately to an ON...THEN... statement, then to a POLL statement, and then to a service routine in the BASIC program. Program control eventually finds its way back to the statement following the WAIT statement and continues sequential execution from that point.

INTERRUPT CONDITIONS

There are four interrupt conditions which a BASIC program can respond to. One of these conditions is caused by the occurrence of a numeric SIZE error during program execution. The four interrupt conditions are as follows:

SRQ (Service Request)

This condition occurs when a peripheral device on the General Purpose Interface Bus requests service by activating the SRQ signal line. Normally, a peripheral does not release the SRQ signal line until it is polled by the processor; if it doesn't get serviced, then it never releases SRQ and the system aborts further operations.

If an SRQ is generated by a peripheral device, refer to that device's Operator's manual for instructions.

NOTE

According to the IEEE GPIB Standard: If several devices are connected to the GPIB bus, one more than 50% of the devices must be turned on (regardless of whether they are actually used), or the GPIB may be loaded down by the turned-off devices, causing a spurious SRQ signal on the bus.

EOI (End Or Identify)

This condition signals the end of a data transfer over the General Purpose Interface Bus and is activated by the source of the transmission as the last byte of data is placed on the bus. The meaning of the EOI signal can be redefined in different applications.

EOF (End Of File)

An End Of File condition occurs when the logical end of a tape file is reached on the internal tape drive. The logical unit number Ø must be specified along with the keyword EOF. For example, the statement ON EOF (Ø) THEN 5ØØ transfers program control to the line 5ØØ when the logical end of the current magnetic tape file is reached. The End Of File condition is specified as EOF (Ø).

SIZE Errors

A SIZE interrupt condition (sometimes called a software interrupt) is generated by numeric overflow conditions as a program executes. In general, SIZE errors are caused by math computations which produce out of range numbers. The numeric range of the system is —1.ØE+3Ø8 to 1.ØE+3Ø8. (This range is graphically illustrated in the explanation of the FUZZ statement in the Environmental Control section.)

The SIZE interrupt feature allows a BASIC program to take different courses of action when a SIZE error occurs. If the ON SIZE THEN... statement is not part of a BASIC program, then SIZE errors are considered fatal errors; fatal errors cause program execution to terminate and the appropriate error message to be printed on the GS display.

THE OFF STATEMENT

Syntax Form:

[Line number] OFF [{ EOF (numeric constant) / EOI / SIZE / SRQ }]

Descriptive Form:

[Line number] OFF [interrupt condition]

Purpose

The OFF statement prevents the current program from responding to the specified interrupt condition after the condition is activated by an ON statement. If an interrupt is not specified as a parameter in the OFF statement, then the program's response to all interrupt conditions is disabled.

Explanation

In some phases of program execution, it is desirable to disable a program's response to one or more interrupt conditions. This is done by executing an OFF statement for those particular conditions. For example, the statement OFF SIZE disables the program's response to SIZE errors; this might be desirable in some cases when it is known that the result of an operation is going to be an out of range number and is to be treated as a fatal error. The program's response to SIZE errors can then be re-enabled by executing an ON SIZE THEN... statement at a later point in the program.

If an OFF SRQ statement is executed, it is essential that a SRQ interrupt not occur until the statement ON SRQ is re-executed; otherwise a fatal error occurs and program execution is aborted.

THE ON ... THEN ... STATEMENT

Syntax Form:

Line number ON { EOF (numeric constant) / EOI / SIZE / SRQ } THEN line number

Descriptive Form:

Line number ON interrupt condition THEN line number

Purpose

The ON...THEN... Statement transfers program control to the specified line number in response to the specified interrupt condition.

Explanation

The ON...THEN... statement is normally placed at the beginning of a program, but doesn't have to be. When the ON...THEN... statement is evaluated, apparent action may not be evident; however, the statement enables the BASIC interpreter to respond to the specified interrupt condition when it occurs. The program normally continues executing in sequence after an ON...THEN... statement is executed, but as soon as the specified interrupt condition occurs, the BASIC interpreter finishes the current statement, then transfers program control back to the ON...THEN... statement. From there program control is transferred to the statement number specified in the ON...THEN... statement. The execution of the first RETURN statement transfers program control back to the main program, to the statement following the point where the main program was interrupted.

The following statements form a typical interrupt service routine:

```
100 ON EOF(0)THEN 140
110 FIND 5
120 INPUT @33: Q$
130 GOTO 120
140 PRINT @33: A,B,C$,D
150 END
```

This program finds the logical end of magnetic tape file 5, then adds one logical record to the end of the file. The program starts by activating the EOF interrupt facility for the internal magnetic tape unit. This tells the BASIC interpreter to be on the lookout for an End of File condition during magnetic tape operations. Line 11Ø positions the magnetic tape read/write head to the beginning of file 5. Line 12Ø then inputs the first logical record and assigns the record to Q$. Program control is then transferred to line 13Ø which transfers control back to line 12Ø. Line 12Ø inputs the second logical record in the file and assigns it to Q$.

Lines 12Ø and 13Ø form an endless loop and are in the program specifically to advance the read/write head through the sequential read data file. Each time a logical record is assigned to Q$, the previous record is overwritten. It doesn't matter though, because the purpose of these two statements is not to read the records; only to advance the tape head to the end of the file.

When the EOF character is reached at the end of the last logical record, program control is transferred to line 1ØØ, then to line 14Ø. Line 14Ø sends the data items assigned to the variables A,B,C$,and D to the magnetic tape as one logical record. This record is added to the end of file 5. The old EOF mark is overwritten by the new record and a new EOF mark is recorded just after the new data. Line 15Ø closes the file and terminates the program.

This program shows how to add data to a half full data file. For complete information on internal magnetic tape operations, refer to the section titled Input/Output Operations.

THE POLL STATEMENT

Syntax Form:

[Line number] POL numeric variable , numeric variable ; primary address [, secondary address] [; primary address [, secondary address]] ...

Descriptive Form:

[Line number] POLL target variable for device identifier , target variable for return status information ; address list

Purpose

The POLL statement causes the BASIC interpreter to serially poll each peripheral device on the General Purpose Interface Bus (GPIB) and determine which device is requesting service. When the device is found, the device sends its status byte to the BASIC interpreter over the GPIB.

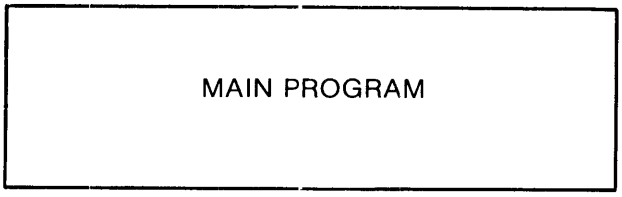
Explanation

The POLL statement is normally executed in response to a service request from a peripheral device on the GPIB. Two numeric variables are specified as parameters in the POLL statement followed by a series of I/O addresses. The BASIC interpreter polls the first I/O address in the list, then the second I/O address, then the third, and so on, until the device requesting service is found. It is imperative that the I/O address of the device requesting service is in the list, or program execution is halted.

After the peripheral device requesting service is found, the device's position in the list is assigned to the first variable specified in the POLL statement. The status word of the device is then sent over the GPIB and assigned to the second variable specified in the POLL statement. After this is accomplished, the program line counter is advanced to the next statement, normally a GOTO...OF... statement, which transfers program control to the service routine for the device requesting service.

The following program is an example of a typical interrupt handling routine:

```
11Ø ON SRQ THEN 197Ø
.
.
```




```
.
.
197Ø POLL M,W;5;12;8,4;3
198Ø GOTO M OF 2ØØØ, 3ØØØ, 4ØØØ, 5ØØØ

2ØØØ REM Service Routine for Device Number 5
.
.
.
299Ø RETURN
3ØØØ REM Service Routine for Device Number 12
.
.
.
399Ø RETURN
4ØØØ REM Service Routine for Device Number 8
.
.
.
499Ø RETURN
5ØØØ REM Service Routine for Device Number 3

.
.
.
599Ø RETURN
.
.
.
7ØØØ END
```

In the beginning of the program, line 11Ø enables the BASIC interpreter to respond to the SRQ (Service Request) interrupt condition; the program then executes in normal sequencial order. If a peripheral signals SRQ while the main program is executing, the BASIC interpreter finishes the present statement and then transfers program control back to line 11Ø — the ON SQR THEN 197Ø statement. This statement then transfers control to the POLL statement in line 197Ø.

The POLL statement in this program contains the numeric variables M and W as parameters followed by I/O addresses 5;12;8,4; and 3. As the BASIC interpreter executes this statement, it first addresses device number 5 to see if it is requesting service; if not, it goes on to device 12, then to device 8, and then to device 3. In this case, secondary address 4 is specified after primary address 8 and is issued immediately after primary address 8. This secondary address might, for example, be used to address a submodule within the mainframe of device 8. If neither of these devices is requesting service, an error occurs. This means another device on the GPIB is requesting service and should have it's I/O address included in the I/O address list.

Assume in this case that device number 12 is requesting service. When the BASIC interpreter polls device 12 and finds it is requesting service, the BASIC interpreter assigns the number 2 to the variable M; a 2 is assigned because device 12 is the second device in the list. If device 3 were requesting service then a 4 would be assigned the the variable M because device 3 is the fourth device in the list.

After a 2 is assigned to the variable M, the BASIC interpreter assigns the status word of device 12 to the variable W — the second variable specified in the POLL statement. This information can be displayed, if desired, by executing a PRINT M,W statement after the POLL statement is executed. The meaning of the status word is device dependent and is defined in the manual for the peripheral device.

After the two variables in the POLL statement have assigned values, the program advances to the next line; in this case, to a GOTO M OF... statement. As explained in the Controlling Program Flow section, this statement transfers program control to one of the line numbers in the line number list. The line number receiving control is specified by the assigned value of M. In this case, M has an assigned value of 2, so program control is transferred to line number 3ØØØ — the beginning of the service routine for device number 12.

When the service routine for device 12 is finished executing, the RETURN statement returns program control back to the main body of the program at the point where the interruption first occurred.

What Happens When Two Devices Request Service at the Same Time?

If two peripheral devices request service at the same time, the first device in the I/O address list gets serviced first. For example, assume that device 12 and device 8 request service at the same time. The BASIC interpreter takes a poll and addresses device 5 followed by device 12. The number 2 is then assigned to the variable M, and program control is transferred to the service routine of device 12 via the GOTO statement in line 198Ø. While this routine is executing, the BASIC interpreter is inhibited from responding to any other SRQ interrupts. Eventually, program control is transferred back to the interruption point in the main program and the BASIC interpreter's response to SRQ is re-enabled. At that time, the program branches back to the POLL statement and executes another serial poll. If devices 5 and 12 are not requesting service, then device 8 is serviced. If, however, device 5 or device 12 requests service in the meantime, they are serviced again. Eventually, when devices 5 and 12 are satisfied, the BASIC interpreter reaches device 8 in the serial poll and the service routine for device 8 is executed.

NOTE

According to the IEEE GPIB Standard: If several devices are connected to the GPIB bus, one more than 50% of the devices must be turned on (regardless of whether they are actually used), or the GPIB may be loaded down by the turned-off devices, causing a spurious SRQ signal on the bus.

Interrupt Service Routines Cannot Be Interrupted

While the BASIC interpreter is responding to a Service Request, it cannot be interrupted to respond to another Service Request. Other peripheral devices may request service, but they can't be serviced until the BASIC interpreter finishes executing the current service routine and branches back to the main program.

THE WAIT STATEMENT

Syntax Form:

[Line number] WAI

Descriptive Form:

[Line number] WAIT

Purpose

The WAIT statement causes program execution to halt and wait for an interrupt condition. This statement ensures that the time taken to begin the interrupt service routine is minimized.

Explanation

The program below is designed specifically to service three external peripheral devices on the General Purpose Interface Bus (GPIB). The WAIT statement causes the BASIC interpreter to wait for one of the peripheral devices to request service.

```
11Ø ON SRQ THEN 14Ø
12Ø WAIT
13Ø GOTO 12Ø
14Ø POLL A,B; 5;1Ø; 15
15Ø GOSUB A OF 2ØØ, 3ØØ, 4ØØ
16Ø RETURN
.
.
2ØØ PRINT "Device 5 is now being serviced"
.
.
29Ø RETURN
3ØØ PRINT "Device 1Ø is now being serviced"
.
.
39Ø RETURN
4ØØ PRINT "Device 15 is now being serviced"
.
.
49Ø RETURN
5ØØ END
```

When line 11Ø is executed, the BASIC interpreter is armed to respond to an SRQ interrupt condition; line 12Ø then causes the BASIC interpreter to wait for an interrupt condition to occur before going to the next statement. If an interrupt condition other than SRQ occurs, the program line counter advances to line 13Ø where it is promptly returned to line 12Ø.

When a peripheral device on the General Purpose Interface Bus activates SRQ, program control is immediately transferred from the WAIT statement in line 12Ø to the ON SRQ statement in line 11Ø and then to line 14Ø where the BASIC interpreter serially polls each device to see which device is requesting service. Once the device is found, its identifying number is assigned to the variable A. (This sequence was just described in the POLL statement). The line counter is then incremented to line 15Ø where the GOSUB statement transfers program control to the device's service routine.

After the service routine is finished executing, the RETURN statement at the end of the routine transfers program control back to line 16Ø, which in turn transfers control back to the interruption point in the main program; in this case, to line 13Ø which returns control to line 12Ø, the WAIT statement. The program then waits for another service request from a peripheral device.

The advantage of the WAIT statement is that it reduces the time it takes the BASIC interpreter to respond to a service request. This time period is called the interrupt latency period.

THE WAIT ROUTINE

Syntax Form:

[Line number] CALL { "WAIT" / string variable } [, numeric variable]

Descriptive Form:

[Line number] CALL routine name [, number of seconds]

NOTE

This routine is not available in the 4051 Graphic System. The delay produced by the WAIT routine is accurate to ±10%.

Purpose

The WAIT routine halts program execution for a specified number of seconds, or until an interrupt condition occurs.

Explanation

The WAIT routine produces a pause in program execution. The number of seconds can be expressed as a constant, variable or expression. If an interrupt condition occurs while a WAIT routine is being processed, the interrupt is handled immediately and no further waiting is done. Interrupt conditions are SRQ, EOI, EOF, and SIZE.

If the numeric expression is non-positive, a zero is assumed (that is, the system pauses for Ø seconds).

If no numeric expression is entered, the routine operates like the WAIT statement.

The following example illustrates how the WAIT routine can be used:

```
68Ø-84Ø (output subroutine to display information)
85Ø CALL "WAIT", 1Ø
86Ø PAGE
87Ø RETURN
```

Lines 68Ø through 84Ø are PRINT, MOVE, and DRAW statements to place a graph or table on the display. Line 85Ø pauses for about 1Ø seconds to let you view the display or decide to make a hard copy. Line 86Ø then erases the screen and line 87Ø returns control to the main program.

INPUT/OUTPUT OPERATIONS

Section 7

INPUT/OUTPUT OPERATIONS

INTRODUCTION TO INPUT/OUTPUT OPERATIONS

System Architecture

The central controller for the Graphic System is supported by the Random Access Memory and the Read Only Memory. All other modules are considered peripheral devices. This includes internal peripheral devices like the GS keyboard and the GS display as well as external devices like a hard copy unit, a Joystick, or an X-Y plotter connected to the General Purpose Interface Bus.

I/O Addressing Facility

The Graphic System has a unique I/O addressing facility which allows each peripheral device in the system to be treated on an equal basis. This includes internal peripheral devices, as well as external peripheral devices.

Each peripheral device in the system is given a peripheral device number called a primary address. Specifying a primary address in a BASIC statement selects a peripheral device for an I/O operation. For example, the statement PRINT @32: selects the GS display as the output device for the PRINT operation; the statement PRINT @33: selects the internal magnetic tape unit as the output device for the print operation; and the statement PRINT @16: selects peripheral device number 16 on the General Purpose Interface Bus as the output device for the print operation.

Each primary address carries with it a second part called a secondary address. The secondary address tells the peripheral device what the I/O operation is all about. For example, the secondary address 12 is issued with each PRINT statement. This secondary address tells the selected peripheral device that the BASIC interpreter is executing a PRINT statement and to prepare to receive information in ASCII code format. The I/O address is issued first before the data transfer begins. After the I/O address is issued, the data transfer takes place. In this case, the selected peripheral device receives the information in ASCII code format, then prints the information on its recording media.

In most cases, the I/O address entry in a BASIC statement is optional. If an I/O address is not specified, the BASIC interpreter automatically issues an I/O address appropriate for the keyword.

Graphic System Keyboard

The Graphic System keyboard is the primary input device for the system. While the system is idle, entries from the keyboard are either evaluated immediately or placed in memory for later use. While the system is operating under program control, the INPUT statement is used to make keyboard entries. The INPUT statement allows numeric data, array elements, and character strings to be input into memory from the GS keyboard while the system is operating under program control. When an INPUT statement is executed, program execution halts and a blinking question mark appears on the GS display. After a keyboard entry is made and the RETURN key is pressed, program execution continues in normal sequential order. Normally, a PRINT statement is executed prior to an INPUT statement to print a message on the GS display. The message tells the keyboard operator what to enter .

Graphic System Display

Printing Data Items

The Graphic System display is the primary output device. Each keyboard entry is printed on the GS display. Data items specified in PRINT statements are also printed on the GS display. The PRINT statement also sends numeric data, array elements, and character strings to the GS display for viewing while the system is operating under program control. The information is printed according to the guidelines specified in a default print format, unless a different print format is specified via a PRINT USING statement. The system offers virtually an unlimited choice of print formats. Each print format is specified in an IMAGE statement.

Listing a BASIC Program

The BASIC program currently in memory is listed on the GS display by executing a LIST statement. One line in the program can be listed, a small portion of the program can be listed, or the entire program can be listed. If an I/O address is specified in a list statement, the program listing is sent to the specified peripheral device in ASCII code format.

An Internal Data File

The DATA statement in a BASIC program acts like an internal data file for storing numeric data and character strings. The data items in the DATA statement are assigned to variables with the READ statement as the program executes.

Magnetic Tape Operations

Creating Files On Magnetic Tape

Before information can be stored on magnetic tape, empty files must first be created. New files are created on the magnetic tape with the MARK statement. Each file can be any given length. Once a file is created, the file can be used to store a BASIC program in ASCII format, or data in either ASCII format or binary format.

Finding Files on Magnetic Tape

The magnetic tape head is positioned to the beginning of a tape file by executing a FIND statement. The FIND statement not only finds the file, but it opens the file for access. This is analogous to opening a disc file for access.

Saving a BASIC Program on Magnetic Tape

The BASIC program currently in memory can be saved on magnetic tape by executing a FIND statement to open a file, then executing a SAVE or CALL "BSAVE" statement. The file must be large enough to hold the BASIC program. The SPACE function is normally used to find out how large a BASIC program is before the program is sent to the magnetic tape. The BASIC program is stored in ASCII code format with the SAVE command, and in binary format with the "BSAVE" routine.

Saving a BASIC Program in Secret Format

The current BASIC program can be saved on magnetic tape in a secret format by executing a FIND statement, a SECRET statement, then a SAVE statement. The program is still stored in ASCII code, but the format is scrambled and only the Graphic System has the ability to unscramble the format when the program is brought back into memory.

Recovering a BASIC Program from Magnetic Tape

A BASIC program stored on magnetic tape is brought back into memory by executing a FIND statement, then an OLD, CALL "BOLD ', or CALL "LINK" statement. These statements clear the entire contents of the Random Access Memory before the BASIC program is brought in from the magnetic tape except that all variable dimensions and assignments are retained by the "LINK" routine. If the BASIC program is marked SECRET, the program can only be executed; the program can never be listed, saved or output from the machine. Secret programs are removed from memory be executing a DELETE, OLD, CALL "BOLD", or CALL "LINK" statement, or by turning off the system power. ASCII programs are brought into memory by the OLD statement; binary programs by the "BOLD" routine.

Appending Programs from Magnetic Tape

Program lines stored on magnetic tape can be added to the BASIC program currently in memory by executing an APPEND or CALL "BAPPEN" statement. A dummy statement in the current BASIC program acts as a target for the incoming program lines. If the dummy statement is specified at the end of the current BASIC program, then the program lines coming in are added to the end of the program. If the dummy statement is specified in the middle of the current BASIC program, the incoming program lines are inserted into the middle of the

program at the specified point. The program lines beyond that point are moved down to make room for the new lines coming in. The newly appended program lines and the lines that are moved down are automatically renumbered with a specified line number increment. ASCII programs are appended to memory with the APPEND command; binary programs with the "BAPPEN" routine.

Storing DATA in ASCII code format on Magnetic Tape

Numeric data and character strings are stored in a magnetic tape file by executing a FIND statement, then a PRINT @33: statement. Primary address 33 specifies that the data is to be sent to the internal magnetic tape.

All of the data items specified in a PRINT statement are treated as a single unit called a logical record. When the data is brought back into memory with the INPUT statement, one logical record is brought back in at a time. If an ASCII data file is partially filled, more data can be added to the file. The magnetic tape head must first be positioned to the EOF (End Of File) mark in the file. This is done by executing an INPUT statement for each logical record in the file; when the EOF mark is found, more data items can be added until the physical end of the file is reached.

Recovering ASCII Data from Magnetic Tape

The INPUT statement is used to recover ASCII data items from the magnetic tape. The FIND statement must be executed first to open the file for access. The INPUT statement is then used to bring in the ASCII data, one logical record at a time.

Storing Data in Binary Format on Magnetic Tape

Numeric data and character strings are stored in machine dependent binary code by executing a FIND statement, then a WRITE statement. The term "machine dependent binary code" refers to the internal format used by the Graphic System to store data in the Random Access Memory. Binary data transfers are normally faster and require less storage space, because the conversion back and forth to ASCII code is eliminated.

Recovering Binary Data from Magnetic Tape

Data stored in binary format is brought back into memory by executing a FIND statement, then a READ @33: statement. The variables specified in the READ statement must match the data item type in the binary file; that is, a numeric variable must be specified if a numeric data item is the next item in the file; likewise, a string variable must be specified if the next data item is a character string. If the data item type is unknown, the TYP function can be executed to determine the data item type before a READ statement is executed.

Closing a Magnetic Tape File

Closing a magnetic tape file is an important step after a PRINT, or WRITE operation. A magnetic tape file can be closed by executing a FIND statement, a CLOSE statement, or an END statement. Closing a file makes sure that the last pieces of information remaining in the magnetic tape memory buffer are forced onto the magnetic tape before the system power is turned off. Closing a magnetic tape file after an OLD, INPUT, or READ operation is not necessary although it is good programming practice to do so.

Killing a Magnetic Tape File

An old magnetic tape file can be reused to store new information if the file is first killed with the KILL statement. When a KILL statement is executed, a fast search for the file is initiated. When the file is found, the file header is marked NEW and the old information in the file cannot be recovered unless the magnetic tape status is changed to non-header format. The file can then be used to store either a new BASIC program, or data in either ASCII or binary format. Although the information in a program file and an ASCII data file can be overwritten with new ASCII data without killing the first file, it is good practice to kill the file first. Likewise, information in a binary data file can be overwritten with new binary data without killing the file first, but again, it is good practice to kill the file first. This is the only way to change a file tape from ASCII to binary and vice versa.

Listing a Magnetic Tape Directory on the GS Display

If the contents of a magnetic tape file are unknown, a TLIST statement can be executed to list the contents of the tape cartridge on the GS display. Executing TLIST rewinds the current magnetic tape to the beginning. The information stored in each file header is then printed on the GS display. This information includes the file number, the file type (program or data), the data storage format (ASCII or binary), the program storage format (secret or non-secret), and the maximum storage capacity of the file in bytes. If an I/O address is specified in the TLIST statement, the tape directory is sent to the specified external peripheral device.

External Peripheral Devices on the GPIB

If the appropriate primary address is specified in the statements just mentioned, I/O operations can be carried on with any peripheral device in the system over the General Purpose Interface Bus. The SAVE or CALL "BSAVE" statement sends a copy of the current BASIC program to the specified peripheral device over the GPIB. The OLD, CALL "BOLD" or CALL "LINK" statement brings in a BASIC program back from the specified peripheral device over the GPIB and places the program in memory. Data transfers are carried on with an external peripheral device just like they are with the internal magnetic tape unit. The PRINT statement sends data to the specified peripheral device in ASCII code format. The INPUT statement receives data in ASCII code format.

Likewise, the WRITE statement sends data to a specified peripheral device in machine dependent binary code. The READ statement is used to bring the data back and place it in memory. The only difference in data transfers to and from the internal magnetic tape and data transfers to and from an external peripheral device is the primary address specified after the keyword.

Direct Access to the General Purpose Interface Bus

Quite often it is desirable to connect a peripheral device to the General Purpose Interface Bus which doesn't have the ability to talk in ASCII code format or the machine dependent binary format used by the Graphic System. In cases like this, the WBYTE (Write Byte) and RBYTE (Read Byte) statements can be used to communicate with the peripheral device. These two statements give direct access to the General Purpose Interface Bus providing the capability to transmit and receive any eight bit binary pattern over the bus. This includes primary talk addresses, primary listen addresses, and universal controller commands, as well as data bytes. Although this method of interfacing is slow and primitive, it provides the Graphic System with an almost unlimited capability to interface to the outside world.

INPUT/OUTPUT (I/O) ADDRESSES

Syntax Form:

{%, @} numeric expression [, numeric expression] :

Descriptive Form:

{%, @} primary address [, secondary address] :

Purpose

The I/O address in a BASIC statement specifies which peripheral device is to take part in the I/O operation. The I/O address also tells the peripheral device what the I/O operation is all about.

Explanation

The Graphic System Input/Output Facility—An Overview on How It Works

When a data transfer takes place within the Graphic System, information is either transferred from a peripheral device to the Random Access Memory, or from the Random Access Memory to a peripheral device. For example, the PRINT statement takes information stored in the Random Access Memory and sends the information to a peripheral device in ASCII code format. The PRINT statement counterpart, INPUT, receives information from a peripheral device in ASCII code format and stores the information in the Random Access Memory. In every I/O statement (except WBYTE and RBYTE) the data transfer occurs between only one peripheral device and the Random Access Memory. Data transfers between two peripheral devices on the General Purpose Interface Bus are set up with the WBYTE (Write Byte) statement.

The Modular Design of Each I/O Statement

Each I/O statement in the Graphic System BASIC language can be subdivided into four parts as shown below:

Line Number KEYWORD I/O address parameters

Each part plays a specific role in the execution of the statement. The purpose of each part is described as follows.

Line Number. The line number determines when the statement is executed. If a line number is present, the statement is executed when the system is placed under program control. If the line number is not present, the statement is executed as soon as the statement is entered into the line buffer and the RETURN key is pressed.

KEYWORD. The KEYWORD is an alphabetical code which tells the BASIC interpreter what function to perform. This code represents a set of instructions for the BASIC interpreter only and is never seen by the peripheral device involved in the transfer. A list of the instructions the BASIC interpreter follows for each keyword can be found in Appendix B.

I/O Address. The I/O address is a two-part numeric code which represents instructions to the peripheral device. The I/O address is sent to the peripheral device before the data transfer begins.

The I/O address follows the keyword in the statement and is specified as an "at" sign (@) or a percent sign (%), followed by a primary address, followed by a comma, followed by a secondary address, and terminated with a colon (:).

The "at" sign (@) or the percent sign (%) specifies which delimiters are to be used during the I/O operation. (Refer to the topic Processor Status in the Environmental Control section for details.)

The primary address is specified as a peripheral device number between 1 and 255. When the statement is executed, the peripheral device number is converted to a primary talk address or a primary listen address, whichever is appropriate for the keyword, and issued to the specified peripheral device. The primary address tells the peripheral device that it has been selected to either send data to or receive data from the random access memory. Peripheral device numbers for the system are divided into categories as follows:

Device Number	Peripheral Device
1-3Ø	External peripheral devices on the General Purpose Interface Bus
31-8Ø	Internal peripheral devices on the General Purpose Interface Bus
81-255	Reserved for future use

Internal peripheral devices are preassigned the following peripheral device numbers:

Device Number	Peripheral Device
31	GS keyboard
32	GS display
33	Magnetic Tape Unit
34	DATA Statement
35-36	Unassigned
37	Processor Status
38-40	Unassigned
41	Left-most ROM slot
42-50	Unassigned
51	2nd-from-left ROM slot
52-60	Unassigned
61	3rd-from-left ROM slot
62-70	Unassigned
71	4th-from-left ROM slot
72-80	Unassigned

NOTE

When referring to the left-most, 2nd-from-left, etc., ROM slot in the preceding table, "left" is from the user's point of view; that is, in front of the keyboard and facing the display.

Peripheral device numbers can be specified as a numeric expression in a statement as long as the BASIC interpreter can reduce the expression to a numeric constant and round the constant to an integer within the range 1 to 255. This means that the primary address can be specified as a numeric variable; by changing the value assigned to the variable, different peripheral devices can be selected as the input source or output destination without changing the BASIC statement itself.

The secondary address in an I/O address is issued immediately after the primary address and tells the peripheral device what the data transfer is all about. Since the peripheral device never sees the keyword in the statement, the secondary address provides the only way to tell the peripheral device what function is being performed by the BASIC interpreter. A secondary address is specified as a number from Ø through 32. Each number has a predefined meaning. For example, secondary address 12 means that the BASIC interpreter is executing a PRINT statement; secondary address 13 means that the BASIC interpreter is executing an INPUT statement, and secondary address Ø means that the BASIC interpreter is sending status information. The following table lists the secondary address assignments for each I/O function performed by the BASIC interpreter. A list of the instructions a peripheral device should follow when it receives a particular secondary address is given in Appendix B.

GPIB Secondary Addresses										
Secondary Address	Predefined Meaning	Decimal Value	Data Bus							
			8	7	6	5	4	3	2	1
Ø	"STATUS"	96	0	1	1	0	0	0	0	0
1	SAVE	97	0	1	1	0	0	0	0	1
2	CLOSE	98	0	1	1	0	0	0	1	0
3	OPEN	99	0	1	1	0	0	0	1	1
4	OLD/APPEND	1ØØ	0	1	1	0	0	1	0	0
5	CREATE	1Ø1	0	1	1	0	0	1	0	1
6	TYPE	1Ø2	0	1	1	0	0	1	1	0
7	KILL	1Ø3	0	1	1	0	0	1	1	1
8	UNIT	1Ø4	0	1	1	0	1	0	0	0
9	DIRECTORY	1Ø5	0	1	1	0	1	0	0	1
1Ø	COPY	1Ø6	0	1	1	0	1	0	1	0
11	RELABEL	1Ø7	0	1	1	0	1	0	1	1
12	PRINT	1Ø8	0	1	1	0	1	1	0	0
13	INPUT	1Ø9	0	1	1	0	1	1	0	1
14	READ	11Ø	0	1	1	0	1	1	1	0
15	WRITE	111	0	1	1	0	1	1	1	1
16	ASSIGN	112	0	1	1	1	0	0	0	0
17	"ALPHASCALE"	113	0	1	1	1	0	0	0	1
18	FONT	114	0	1	1	1	0	0	1	0
19	LIST/TLIST	115	0	1	1	1	0	0	1	1
2Ø	DRAW/RDRAW	116	0	1	1	1	0	1	0	0
21	MOVE/RMOVE	117	0	1	1	1	0	1	0	1
22	PAGE	118	0	1	1	1	0	1	1	0
23	HOME	119	0	1	1	1	0	1	1	1
24	GIN	12Ø	0	1	1	1	1	0	0	0
25	"ALPHAROTATE"	121	0	1	1	1	1	0	0	1
26	COMMAND	122	0	1	1	1	1	0	1	0
27	FIND	123	0	1	1	1	1	0	1	1
28	MARK	124	0	1	1	1	1	1	0	0
29	SECRET	125	0	1	1	1	1	1	0	1
3Ø	"ERROR"	126	0	1	1	1	1	1	1	0
31	undefined	127	0	1	1	1	1	1	1	1

A secondary address can be specified as a numeric expression as long as the numeric expression can be reduced to a numeric constant and rounded to an integer within the range Ø to 32. If 32 is specified as a secondary address, the BASIC interpreter is inhibited from issuing a secondary address.

The colon (:) specified after the secondary address acts as the I/O address delimiter.

Parameters. The parameters of a BASIC statement are specified after the colon in the I/O address. The parameters of a statement represent the information to be transferred. For example, in the statement FIND @33,27:5, the tape file number 5 is the data transferred from the random access memory to the specified peripheral device after the I/O address is issued.

Statement Execution

When the statement FIND @33,27:5 is executed, the I/O address @33,27: is issued to the internal magnetic tape unit. Peripheral device number 33 is converted to the primary listen address for the internal magnetic tape and tells the tape unit to prepare to receive an ASCII character string. Secondary address 27 tells the magnetic tape unit that the ASCII character string represents a tape file number to be found. The number 5 is then converted to an ASCII character string and sent to the internal magnetic tape. The magnetic unit reads the number 5 and executes a fast search to the beginning of file 5.

It is interesting to note here that the internal magnetic tape is totally dependent on the secondary address for the meaning of the I/O transfer. For example, if the statement FIND @33,7:5 is executed, the parameter 5 is still sent to the internal magnetic tape unit as an ASCII character string; this time, however, the secondary address 7 tells the internal tape that the parameter 5 represents a tape file to be killed rather than a tape file to be found. This statement is the same as executing a KILL @33,7:5 statement. As another example, the statement FIND @33,12:5 causes the internal magnetic tape to record the parameter 5 in ASCII code format starting at the present position of the read/write head, because the secondary address 12 makes the tape unit think that the BASIC interpreter is executing a PRINT statement. This statement produces the same result as executing a PRINT @33,12:5 statement.

It is apparent from the above discussion that all a keyword does is tell the BASIC interpreter to convert the specified parameters into an ASCII character string, issue the specified I/O address, then issue the ASCII character string. The BASIC interpreter really never knows (or cares) where the ASCII character string goes or how it is interpreted. From the peripheral's point of view, it never sees the keyword in a statement or what the BASIC interpreter is doing. All it can assume is that the BASIC interpreter is executing the function described by the secondary address and interpret the ASCII data string as the parameters of that function.

Default I/O Address

In most cases, specifying an I/O address in a BASIC statement is optional. If an I/O address is not specified, the BASIC interpreter inserts an I/O address appropriate for the keyword. This I/O address is called the default I/O address for the keyword. For example:

```
FIND 5
```

When this statement is executed, the BASIC interpreter automatically inserts the I/O address @33,27: into the statement. The result is shown below:

```
FIND @33,27:5
```

This default I/O address selects the internal magnetic tape as the peripheral device to receive the FIND information. The following table lists the default I/O address for each keyword in the language.

DEFAULT I/O ADDRESSES

APPEND	@ 33,4:
BRIGHTNESS	@ 32,30:
CHARSIZE	@ 32,17:
CLOSE	@ 33,2:
COPY	@ 32,10:
DASH	@ 32,31:
DRAW	@ 32,20:
FIND	@ 33,27:
FONT	@ 32,18:
GIN	@ 32,24:
HOME	@ 32,23:
INPUT	@ 31,13:
KILL	@ 33,7:
LIST	@ 32,19:
MARK	@ 33,28:
MOVE	@ 32,21:
OLD	@ 33,4:
PAGE	@ 32,22:
PRINT	@ 32,12:
RDRAW	@ 32,20:
READ	@ 34,14:
RMOVE	@ 32,21:
SAVE	@ 33,1:
SECRET	@ 37,29:
TLIST	@ 32,19:
WRITE	@ 33,15:

If a primary address and/or secondary address is specified in an I/O statement, the specified address is issued instead of the default address. For example:

```
FIND @22:5
```

When this statement is executed, the BASIC interpreter issues the primary listen address for device 22 instead of the default primary listen address for device 33. This primary address selects peripheral device number 22 on the General Purpose Interface Bus to receive the FIND information. Because a secondary address is not specified in this case, the BASIC interpreter automatically issues 27 as a default secondary address. The parameter 5 is then converted to an ASCII character string and sent to device 22 over the GPIB.

It is up to device 22 to interpret the secondary address 27 as meaning the BASIC interpreter is sending the parameter of a FIND statement and then find the specified file.

If a secondary address is specified as well as a primary address, then the specified secondary address is issued instead of the default secondary address. For example:

```
FIND @22,12:5
```

When this statement is executed, the primary listen address for device 22 is issued, followed by the secondary address 12, followed by the parameter 5. In this case, it is up to device 22 to interpret the secondary address 12 as meaning the BASIC interpreter is sending the parameter to a FIND statement.

Care must be taken when specifying a different secondary address for a keyword. In this case, if peripheral device 22 is built to conform to the predefined meanings of the Graphic System secondary addresses, device 22 will PRINT the parameter 5 instead of FIND file 5 because the secondary address 12 is the default secondary address for the PRINT statement.

Duplicating Output Statements with the PRINT statement

Due to the modular design of the I/O addressing facility, virtually every I/O statement involving ASCII data output can be duplicated with the PRINT statement. This includes magnetic tape statements like FIND, MARK, and SAVE as well as display statements like MOVE and DRAW. In most cases, all a peripheral device needs is the proper primary and secondary address and an ASCII data string specifying the parameters to be used in the operation. For example:

```
FIND 5
```

When this statement is executed, the BASIC interpreter inserts the I/O address @33,27: into the statement, converts the number 5 to an ASCII character string, issues the I/O address, then issues the character string.

The important thing to keep in mind here is that the peripheral device never sees the keyword in the statement—only the I/O address and the parameters. The primary function the keyword FIND has is to insert the proper default I/O address if one is not specified and make sure the statement is syntactically correct; that is, the keyword FIND in this case makes sure that only one number is specified as a tape file number and that number falls in the range Ø through 255.

Since the peripheral device never sees the keyword in a statement, the PRINT statement can be used to duplicate the FIND statement. For example:

```
PRINT @33,27:5
```

When this statement is executed, the internal magnetic tape receives the same information as it did with the FIND statement. The BASIC interpreter issues the primary listen address for device 33, followed by the secondary address 27, followed by the parameter 5 in the form of an ASCII character string terminated by a Carriage Return. The internal magnetic tape responds by positioning the tape head to the beginning of a file 5.

NOTE

Intermixing different keywords and secondary addresses in magnetic tape statements on early production units may cause an error message to be printed on the GS display even though there is no error.

Using the PRINT statement to execute a FIND allows a little more freedom in parameter specification. For example, assume that peripheral device 14 is connected to the General Purpose Interface Bus (GPIB) and is designed so that it requires two parameters to execute a FIND function, one for the tape file number and one which specifies the data item in the file to be found. The FIND statement cannot be used in this case because the keyword FIND is limited to one parameter between Ø and 255. The PRINT statement, however, can be used to execute the FIND function for device 14. For example:

```
PRINT @14,27:5,4
```

When this statement is executed, the primary listen address for device 14 is issued over the GPIB, followed by secondary address 27, followed by the parameters 5 and 4 in the form of an ASCII character string. In this case, secondary address 27 tells device number 14 to execute a FIND function with the parameters 5 and 4. The first parameter 5 could mean to find file 5 and the second parameter 4 could mean to position the tape head to the fourth data item in the file. When a PRINT statement is used to execute a FIND function, practically anything can be specified as parameters after the I/O address. This allows the Graphic System to issue parameters which conform to the requirements of practically any peripheral device, whatever they may be.

Executing a PRINT with the Keyword FIND

Just as a FIND statement can be duplicated with a PRINT statement, a PRINT statement (limited to one parameter) can be duplicated with a FIND statement. For example:

```
FIND @32,12:5
```

When this statement is executed, the primary listen address for device 32 (the GS display) is issued, followed by the secondary address 12, followed by the parameter 5 in the form of an ASCII character string terminated by a Carriage Return. Since the secondary address 12 is predefined to mean that the BASIC interpreter is executing a PRINT statement, the GS display prints the 5, then executes a Carriage Return. Although this statement has little practical value, it does illustrate how different primary and secondary address can be combined to execute functions which are totally unrelated to the keyword in the statement. A practical use for this facility is described in the following topic.

Specifying DRAW Coordinates in GDUs Using the PRINT statement

The coordinates of a DRAW statement can be specified directly in GDUs and sent to the GS display or an external peripheral device via the PRINT statement. A DRAW of this type is fast because the transformation from user data units to graphic display units is eliminated. The draw is executed as follows:

```
PRINT @32,2Ø:8Ø,8Ø
```

When this statement is executed, the BASIC interpreter issues the primary address 32 and the secondary address 2Ø. This tells the GS display to execute a DRAW after it receives the X and Y coordinates of a graphic data point. After the I/O address is issued, the parameters (8Ø,8Ø) are converted to an ASCII character string using the default PRINT format and are sent to the GS display. As far as the BASIC interpreter is concerned, the parameters are being sent to the GS display for printing. The GS display, however, interprets the information as the X and Y coordinates of a data point in the GDUs because it received the secondary address 2Ø. Once the information is received, the GS display draws a vector from the present position of the cursor to the coordinates (8Ø,8Ø). The WINDOW and VIEWPORT parameters have no effect on this kind of DRAW.

More than one vector can be drawn in this fashion by specifying more than one pair of coordinate values. For example:

```
PRINT @32,2Ø:8Ø,8Ø,5Ø,3Ø
```

When this statement is executed, the GS display draws two vectors. The first vector is drawn from the present position of the alphanumeric cursor to the coordinates (8Ø,8Ø). The second vector is drawn from the coordinates (8Ø,8Ø) to the coordinates (5Ø,3Ø). Remember, these values are in GDUs.

If an array is specified in the PRINT statement, then the elements in the array are paired off in row major order and used as X-Y coordinates. For example:

```
PRINT @32,2Ø:A
```

When this statement is executed, array A is sent to the GS display as a series of X-Y coordinates. If A has one dimension, then the elements A(1) and A(2) are used as the first X-Y coordinates; the elements A(3) and A(4) are used for the second X-Y coordinates, and so on. If array A has two dimensions, two rows and four columns for example, then the elements A(1,1) and A(1,2) are used as the coordinates for the first vector, the elements A(1,3) and A(1,4) as the coordinates for the second vector, the elements A(2,1) and A(2,2) for the third vector, and so on. Notice that this method of using arrays to draw vectors is different than the method used in the DRAW statement.

THE APPEND STATEMENT

Syntax Form:

[Line number] APP [I/O address] line number [, numeric expression]

Descriptive Form:

[Line number] APPEND [I/O address] target line number in current program

[, increment between line numbers]

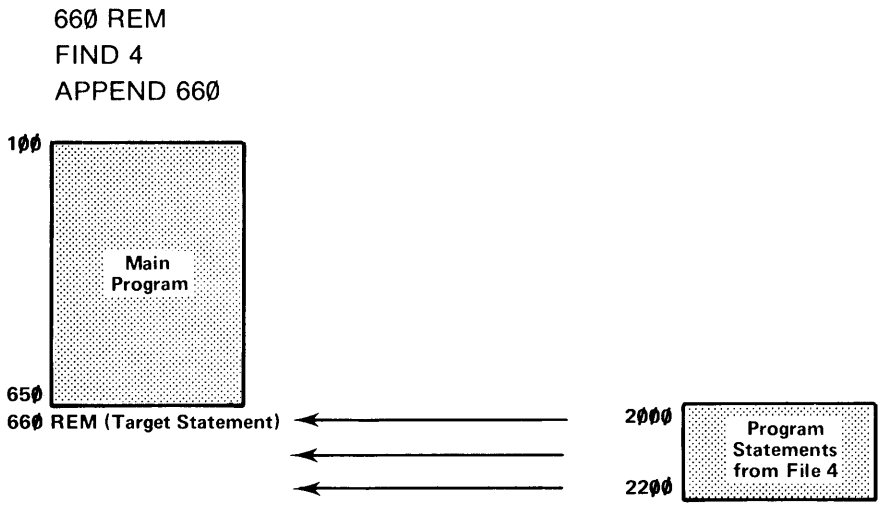
Purpose

The APPEND statement inputs BASIC statements in ASCII format from a specified peripheral device and adds the statements to the BASIC program currently in memory.

Explanation

Before an APPEND statement is executed, the read head of the specified peripheral device is normally positioned at the beginning of a program file (the file containing the statements to be appended). If a peripheral device is not specified, then the internal magnetic tape unit is selected as the peripheral device by default. Next, a statement in the current BASIC program is selected as a target to mark the entry point for the new statements. This target statement is overwritten by the first statement coming in from the peripheral, so it is normally a dummy statement (such as a REMARK statement) created specifically to act as a target for the APPEND operation.

The following figure illustrates a simple APPEND operation. The statements in program file 4 on the internal magnetic tape unit are appended (added) to the end of the BASIC program currently in the Random Access Memory. The following statements are executed to accomplish the task:



Line 66Ø is a dummy REMARK statement created as a target for the incoming statements from file 4. The last statement in the main program can be used as the target statement in this case, but it must be remembered that the target statement is overwritten by the first statement coming in from the magnetic tape; it can not be recovered, so if all the statements in the current program are valid, then a dummy target statement must be created first.

The second statement (FIND 4) positions the internal magnetic tape read/write head to the beginning of file 4. This file must contain valid BASIC statements, or an error occurs when the APPEND statement is executed.

The third statement (APPEND 66Ø) starts the APPEND operation. All of the statements in file 4 are brought into memory. The first statement brought in is given the line number 66Ø and overwrites the target statement. The remaining statements are renumbered, if necessary, from line number 66Ø on, with an increment of 1Ø (the default value). If the APPEND statement specifies an increment (APPEND 66Ø,5 for example) then the BASIC interpreter renumbers the newly appended statements with the specified increment; starting with line number 66Ø in this case, the BASIC interpreter gives the next line the number 665, the next line 67Ø, and so on.

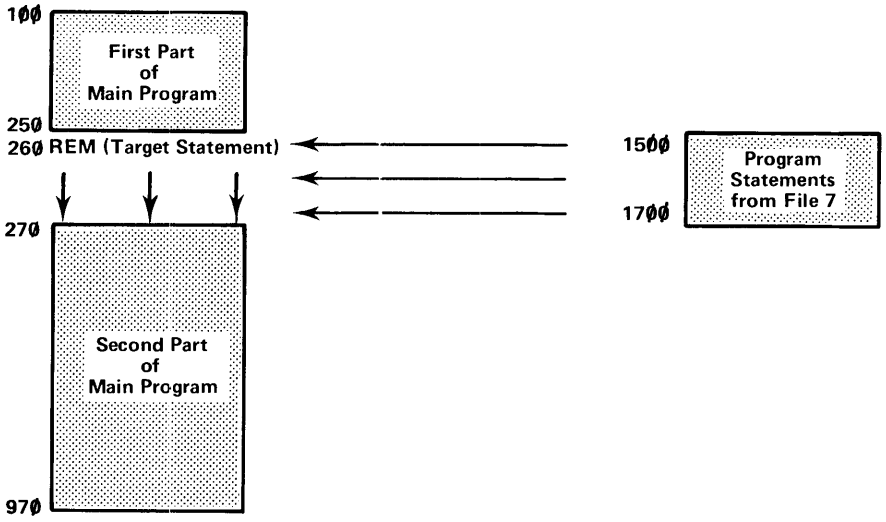
Inserting Statements into a Program

The APPEND statement can also be used to insert statements into the middle of a BASIC program. The operation is the same as adding statements to the end. A dummy statement marks the entry point. When the APPEND statement is executed, all the statements beyond the

dummy statement are moved down to make room for the new statements coming in. At the end of the operation, the newly appended statements and the statements beyond them are renumbered according to the specified increment. If an increment is not specified, then the default increment (10) is used.

The following figure illustrates an insertion operation:

```
260 REM
FIND 7
APPEND 260,100
```



In this figure, the BASIC program statements stored in file 7 are inserted into the main program starting at line 260. To do this, a dummy target statement is created in line 260 to mark the entry point. A statement currently in the program can be selected as the target, but this is usually undesirable because the target statement is overwritten and can't be recovered. The read head of the internal magnetic tape unit is then positioned at the beginning of file 7 with a FIND 7 statement; the APPEND statement is executed next to start the operation.

In this example, all of the statements in file 7 are brought into the memory. The statements beyond line 260 in the main program are moved down to make room for the new statements coming in. The newly appended statements and the statements which were moved down are then renumbered starting with line number 260 and increase with an increment of 100.

Appending Statements from an External Peripheral Device

Program statements stored in an external peripheral device can be appended to the current program in the same manner as statements stored on the internal magnetic tape. The only difference in the APPEND statement is the addition of an I/O address. For example:

```
APPEND @15:55Ø,2Ø
```

In this statement, device number 15 on the General Purpose Interface Bus (GPIB) is specified as the source of the statements to be appended. Line number 55Ø is specified as the target statement in the current program and the number 2Ø is specified as the renumber increment. The readhead on peripheral 15 must be positioned to the beginning of the desired program file before this APPEND statement is executed; if not an error occurs.

When the APPEND statement is executed, primary address 15 is sent over the GPIB to tell device number 15 that it has been selected for the upcoming data transfer. The secondary address 4 is then issued by default; this tells device 15 to send the contents of the program file presently positioned under its readhead.

After peripheral 15 sends the new program statements to the BASIC interpreter over the GPIB, the BASIC interpreter assigns the first statement to line 55Ø. This overwrites the target statement. The rest of the statements which follow are inserted and renumbered with an increment of 2Ø. This includes any statements in the original BASIC program which were moved down to make room for the appended statements.

WARNING

If two BASIC statements are brought into memory with the APPEND statement, and both statements have the same line number, then unpredictable results will occur when the program is run.

NOTE

Any file APPENDed to a program which is SECRET will be treated as if it were SECRET.

THE BAPPEN ROUTINE

Syntax Form:

[Line number] CALL { "BAPPEN," / string variable, } [I/O address;] line number [, increment]

Descriptive Form:

[Line number] CALL routine name, [I/O address;] target line number in current program [line number / , increment]

Purpose

The BAPPEN (Binary APPEND) routine inputs BASIC statements stored in binary format on the specified peripheral device and attaches those statements to the program currently in memory.

Explanation

The BAPPEN routine follows the same procedure as the APPEND statement. First, the read/write head of the input device must be positioned to the beginning of a binary program file by using the FIND command. Then, when the BAPPEN routine is executed, the given target statement is overwritten by the first statement coming from the peripheral device.

The internal magnetic tape unit is selected by default if a peripheral device is not specified. The newly appended statements and any statements that originally followed the target statement are renumbered, starting with the target line number. The line numbers are incremented by either the given increment, or by the default of 10 if an increment is not specified.

For example:

```
.
.
.
200 REM APPEND STATEMENTS FROM FILE 2 HERE
    FIND 2
    CALL "BAPPEN",200
```

When these statements are executed, the FIND statement positions the read/write head at the beginning of file 2. File 2 must be a binary program file. (Error message number 55 is printed if file 2 is not binary.) The BAPPEN routine replaces line 2ØØ with the first statement in file 2. The remaining statements in file 2 are then appended with line numbers incremented by 1Ø.

If the program you want to append is stored as a SECRET BINARY program, after executing the BAPPEN routine, the entire contents of memory becomes secret. This includes all original statements and the statements appended.

Specifying A Peripheral Input Device

Any peripheral tape drive storing a binary program can be specified as the input device. The device is specified like this:

```
FIND @ 15:3
CALL "BAPPEN",15;650,50
```

This statement selects file number 3 on device number 15 on the General Purpose Interface Bus as the input device. Line number 65Ø is the target statement in the current program stored in memory. The renumber increment is 5Ø.

THE BOLD ROUTINE

Syntax Form:

[Line number] CALL { "BOLD" / string variable } [, I/O address]

Descriptive Form:

[Line number] CALL routine name [, I/O address]

Purpose

The BOLD (Binary OLD) routine copies the program stored in binary format into the memory from the specified input source.

Explanation

Any program that is in binary format can be loaded into memory by using the BOLD routine.

If an input source is not specified, the internal magnetic tape unit is chosen by default.

To retrieve a binary program from the internal magnetic tape drive, the read/write head of the output device must first be positioned at the beginning of a binary program file. For example, the statements:

```
FIND 1
CALL "BOLD"
```

load the binary program from file 1 into memory. The FIND statement locates the beginning of file 1. The BOLD routine erases everything currently in memory, then transfers the binary file into memory. The loaded program is ready to be executed by a RUN statement or edited. Like the OLD command, if the BOLD routine is executed under program control, a RUN statement is automatically executed after the program is loaded into memory.

Specifying A Peripheral Device

Any peripheral tape drive holding a binary program can be specified as the input source for the BOLD routine by designating the appropriate I/O address. The following statement specifies file number 6 on device number 2 on the General Purpose Interface Bus.

```
290 FIND @ 2:6
300 CALL "BOLD",2
```

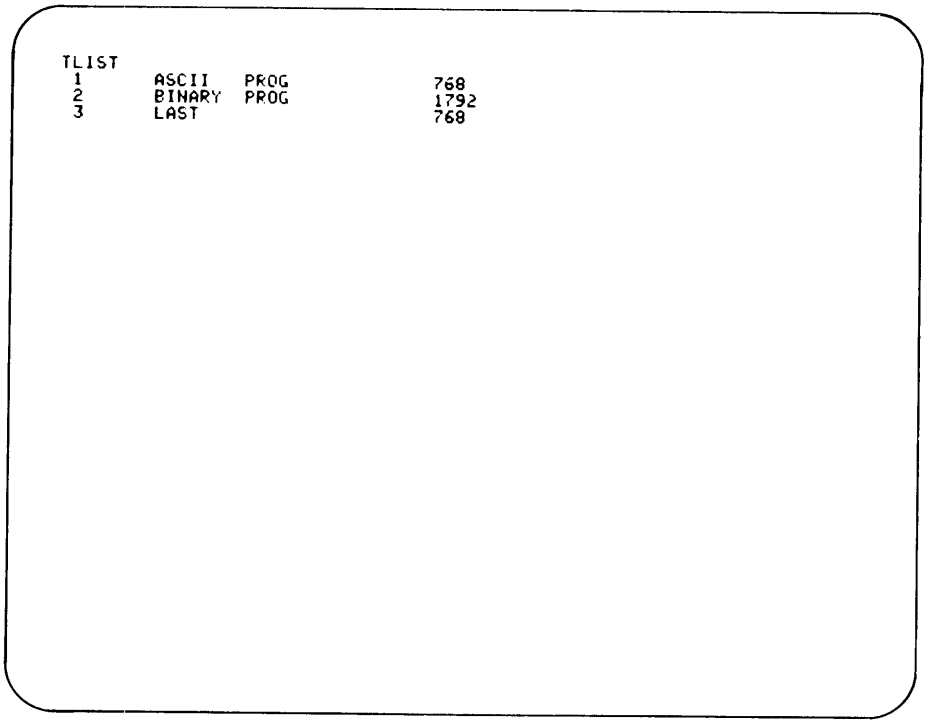
USING AUTO LOAD WITH BINARY FILES

The AUTO LOAD key finds the first ASCII program file on the magnetic tape and then executes an OLD command automatically. The OLD command loads and executes the program. If you want to load the first file with the AUTO LOAD key, the first file must contain an ASCII program.

By storing an ASCII program in file 1 that finds and loads a binary program, you can use the AUTO LOAD key to automatically find and load a binary program. The following example shows how this can be done.

Using a TLIST command, you can see the programs stored on the magnetic tape.

GS Display Output



File 1, an ASCII program file, contains this program:

```
100 FIND 2
110 CALL "BOLD"
```

When the AUTO LOAD key is pressed, the program in file 1 is loaded and executed. This program in turn finds and loads the desired binary program in file 2. Since the CALL "BOLD" statement is executed under program control, the binary program in file 2 is executed after being loaded.

THE BSAVE ROUTINE

Syntax Form:

[Line number] CALL { "BSAVE" / string variable } [I/O address]

Descriptive Form:

[Line number] CALL routine name [, I/O address]

Purpose

The BSAVE (Binary SAVE) routine stores the current BASIC program on the specified output device in binary format.

Explanation

The BSAVE routine sends a copy of the current program to the output device in binary code. Like the SAVE statement, BSAVE does not alter assigned values of variables or system environmental conditions. The CALL "BSAVE" statement can be a step in the program being stored, or it can be executed directly from the Graphic System keyboard.

If an output device is not specified, the internal magnetic tape unit is chosen by default.

To execute the BSAVE routine, the read/write head of the output device must be positioned at the beginning of a file marked BINARY PROG or NEW. The MARK statement must allocate enough space to store the entire program.

Binary programs use more space on magnetic tape than ASCII programs. The amount depends on the size of the file. Because the SPACE function allocates the approximate space for an ASCII program, the SPACE function may or may not allocate enough space to hold the entire program in binary. If you try to execute the BSAVE routine and the selected file is not large enough to hold the current program, error message number 48 is displayed on the screen. To make sure enough space is allocated to hold the current program, use the following method:

Allocate all of the space the program uses in memory. For example, if your Graphic System has 32K bytes of memory space, enter:

```
MARK 1,32000-MEM
```

The MEM function returns the number of bytes still available in memory. By subtracting this amount from the total storage capacity of memory, the remainder is the amount of space the current program occupies in memory. This remainder is also the amount of space needed to store the program on tape.

NOTE

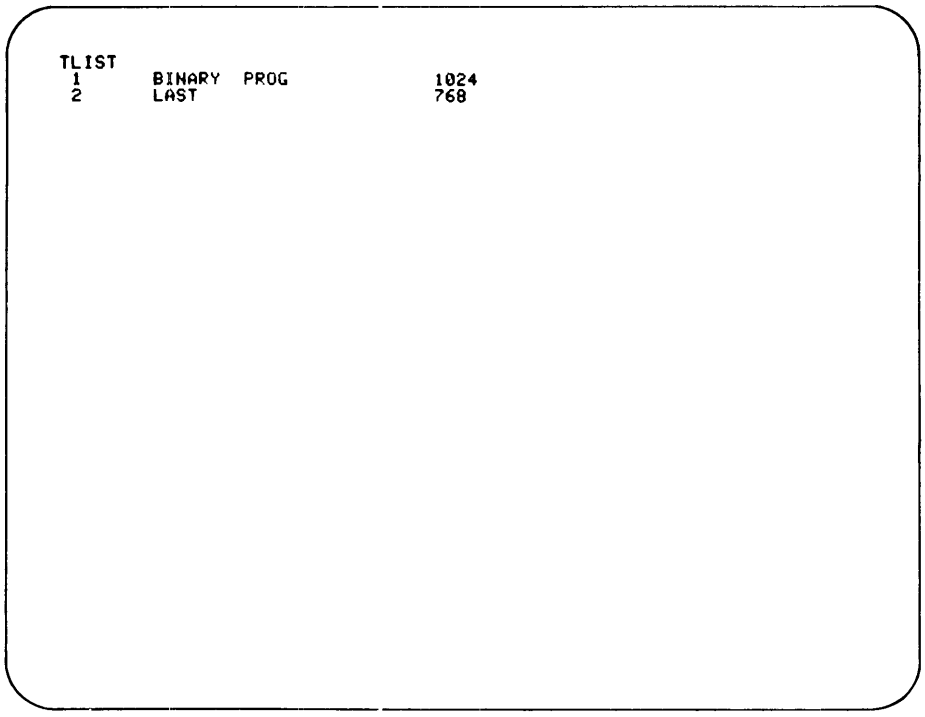
Of the actual storage capacity of Graphic System memory some space is reserved by the processor for a work area. The MEM function only considers the remaining bytes as free. Therefore, the MEM value will be a little less than the actual size of memory.

Using the internal magnetic tape unit by default, the following example stores the current BASIC program in binary format.

```
FIND 0
MARK 1,1000
FIND 1
CALL "BSAVE"
```

Since this is the first file on the tape, the read/write head is positioned at the beginning of the tape by the FIND 0 statement. The MARK statement allocates space on the tape to hold 1000 bytes. The read/write head is then positioned to the start of the allocated space by the FIND 1 statement. The CALL "BSAVE" statement then transfers a binary copy of the current program to file 1.

If you now execute a TLIST statement, you can verify the action performed. Note the header of the internal tape for file 1 is marked BINARY PROGRAM.



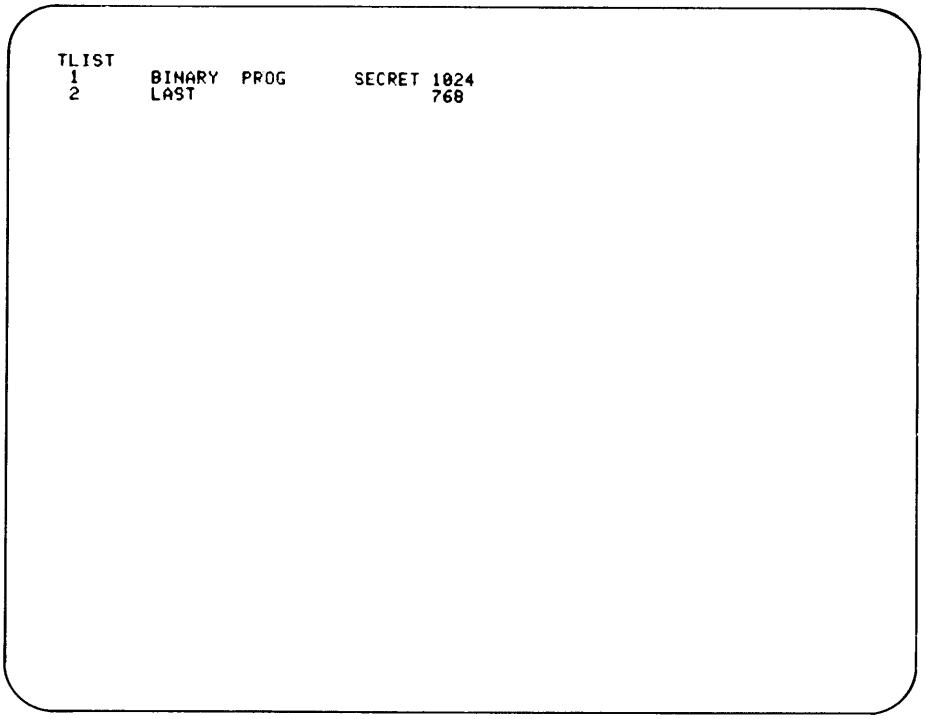
Using SECRET with Binary Programs

The current BASIC program can be marked SECRET and also be stored in binary format. The rules for using SECRET are the same as with any ASCII file. For example:

```
SECRET
FIND 1
CALL "BSAVE"
```

These statements store the current program as a secret program in binary format in file 1. From a TLIST command, you can see that the file header is marked BINARY PROG SECRET.

GS Display Output



To mark the header BINARY PROG SECRET when using an external tape drive, such as the TEKTRONIX 4924 Digital Tape Drive, you must specify the I/O address of the tape drive in a SECRET command. This allows the external device to recognize the program as secret and mark the header accordingly.

For example, if the 4924 Tape Drive is assigned to device number 2 on the General Purpose Interface Bus, the following program stores the current program as secret in binary format.

```
SECRET @ 2:
SECRET
FIND @ 2:1
CALL "BSAVE",2
```

Specifying A Peripheral Device

The current program can be stored in binary format on any peripheral tape drive in the system by specifying the desired primary address in the CALL "BSAVE" statement. For example:

```
290 FIND @ 19:5
300 CALL "BSAVE",19
```

This statement sends the current program in binary format to file number 5 on device 19 on the General Purpose Interface Bus.

CAUTION

Unless the internal tape file is closed when executing a BSAVE to an external device, data may be inadvertently written on the internal magnetic tape. Ensure the internal tape file is closed before executing BSAVE to an external device.

THE CLOSE STATEMENT

Syntax Form:

[Line number] CLO

Descriptive Form:

[Line number] CLOSE

Purpose

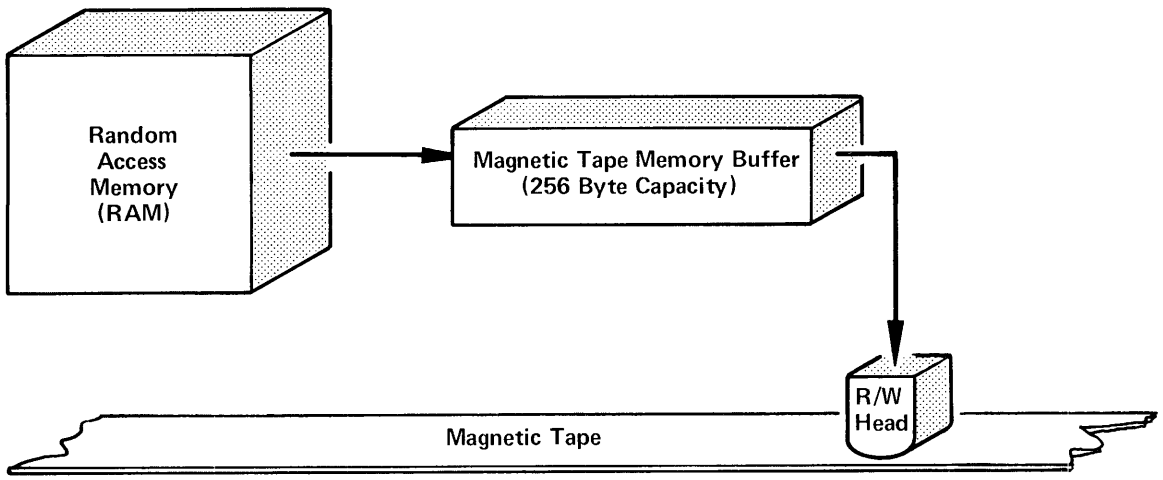
The CLOSE statement closes the current file on the internal magnetic tape unit.

Explanation

The CLOSE statement is used to terminate a PRINT or WRITE operation to the internal magnetic tape. This ensures that any remaining information in the magnetic tape memory buffer is "forced out" or "dumped" from the buffer onto the tape. The CLOSE statement is not needed to terminate a READ, INPUT, or OLD operation.

The Magnetic Tape Memory Buffer

When programs and data are transferred to and from the system memory to the internal magnetic tape unit, the information passes through a 256 byte memory buffer. This memory buffer is shown below.



During PRINT, WRITE, and SAVE operations, information is loaded into the memory buffer until the buffer is full; the buffer is then "dumped" onto the magnetic tape. The buffer is filled and dumped repeatedly until the transfer is complete. (This accounts for the short bursts of tape movement during read/write operations to and from the internal magnetic tape.)

The Need for a CLOSE Statement

Normally, the magnetic tape memory buffer is not dumped onto the magnetic tape until the buffer is full. Quite often, the last few bytes of information only partially fill the buffer. This information remains in the buffer until it is forced out with a CLOSE statement, a FIND statement, or an END statement. If power is removed from the system before executing a CLOSE, FIND, or END statement, then the information in the buffer is lost and cannot be recovered. Pressing BREAK twice also closes the file.

THE DASH STATEMENT

Syntax Form:

[Line number] DAS numeric expression

Descriptive Form:

[Line number] DASH dash pattern

NOTE

This command is not available in the 4051 and 4052 Graphic Systems.

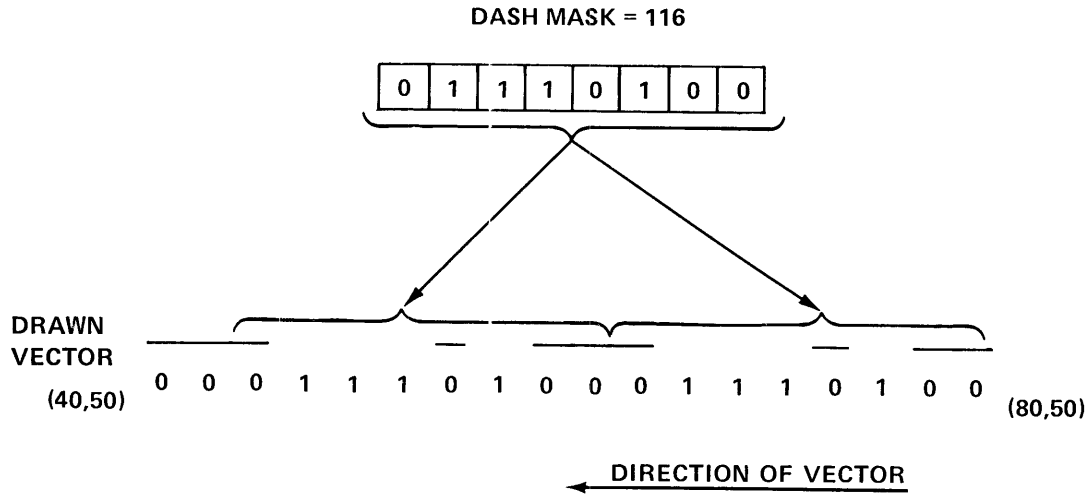
Purpose

The DASH statement specifies how DRAW and RDRAW vectors are displayed: as continuous lines, dashed lines, or dark lines.

Explanation

The dash mask is an integer between Ø and 255 which defines the dash pattern for all DRAW and RDRAW commands on the 4054 Graphic System. To understand how the dash mask works, convert the dash mask number to its binary equivalent. Since the number is between Ø and 255, its binary equivalent will be eight bits (binary digits) long.

Suppose the dash mask is 116 (=Ø111Ø1ØØ in binary), and a vector is drawn from 8Ø,5Ø to 4Ø,5Ø. The vector is drawn as follows:



If a bit is Ø in the dash mask, the corresponding segment is drawn; if a bit is 1, that segment is not drawn. The pattern is repeated as many times as needed to draw the vector, and is not reset the beginning of the pattern after each DRAW or RDRAW.

If the dash mask is Ø (= ØØØØØØØØ), the pattern is a solid line. If the mask is 3 (= ØØØØØØ11), the pattern is a short space followed by a long line. Some more examples follow (remember that the mask is read from right to left):

DASH MASK NUMBER	BINARY EQUIVALENT	PATTERN
3	00000011	———
170	10101010	— — — —
85	01010101	— — — —
15	00001111	——
255	11111111	(no vector is drawn)
200	11001000	— ——

If the mask is greater than 255, 256 is subtracted from the mask until it is between Ø and 255 (256 = ØØØØØØØØ, 257 = ØØØØØØØ1, and so on). If the mask is less than Ø, 256 is added until it is between Ø and 255 (−1 = 11111111, −40 = 11Ø11ØØØ, −255 = ØØØØØØØ1).

The default value for the dash mask is 0. This is reset to 0 by the INIT command. The default address is PRINT @ 32,31:

THE DATA STATEMENT

Syntax Form:

[Line number] DAT { string constant / numeric constant } [, { string constant / numeric constant }] ...

Descriptive Form:

[Line number] DATA data item [, data item] ...

Purpose

The DATA statement stores data items within the BASIC program. These data items can be character strings and/or numeric data and are normally assigned to variables with the READ statement when the program is executed.

Explanation

An Internal Data File

The DATA statement can be thought of as a sequential read data file which is internal to the BASIC program. Data items are "stored" in one or more DATA statements and are assigned to variables using the READ statement. If there is more than one DATA statement in a program, then the DATA statements are linked together in a continuous chain. The DATA statement with the lowest line number is considered the beginning of the internal data file. The end of the first DATA statement is linked to the beginning of the DATA statement with the next highest line number, and so on. The last data item in the highest numbered DATA statement marks the end of the internal data file.

Data Item Pointer

An internal pointer is associated with the DATA statement to indicate which data item is to be read next. The pointer is set to the first data item in the first DATA statement on system power up, after the execution of an INIT statement, a RESTORE statement, or a RUN statement when a line number is not specified as a parameter. After a data item is read with the READ statement, the pointer moves to the next data item to the right.

When the last data item in a DATA statement is read, the pointer moves to the first data item in the next DATA statement, and so on. When the last data item in the last DATA statement is read, the pointer points out into space and must be reset before another READ operation is attempted. The pointer is reset to the beginning of a particular DATA statement with the RESTORE statement, or to the beginning of the lowest line numbered DATA statement with the INIT statement or RESTORE statement

Reading Data Items in a DATA Statement

The following program illustrates how data items can be arranged in a DATA statement and how those items are assigned to variables with the READ statement:

```
100 INIT
110 DATA "Sally",5,"1305 S.W. Henry St."
120 DATA 6.95, "Billy",6,"1501 S.E. Morrison",5.87
130 FOR I=1 TO 2
140 READ A$,B,C$,D
150 PRINT USING 180: A$,B
160 PRINT USING 190: C$,D
170 NEXT I
180 IMAGE "STUDENT NAME:",2X,10A,/,"AGE:",2X,FD
190 IMAGE "ADDRESS: ',2X,30A,/,"ART SUPPLIES:",2X,$+FD.FD,3/
200 END
```

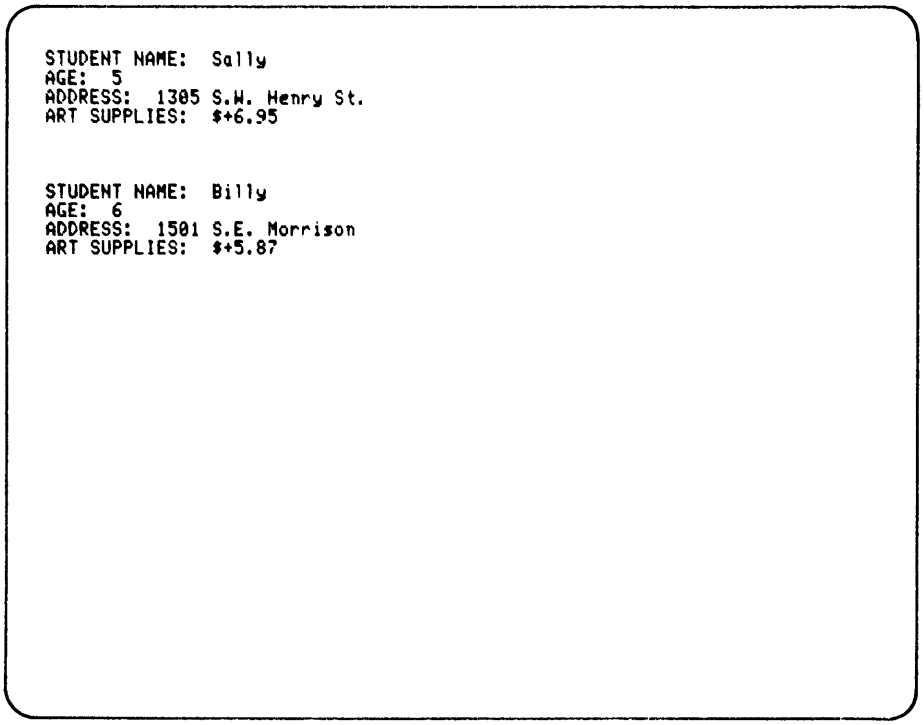
When line 100 in this program is executed, the system parameters are set to their default values and the DATA statement pointer is set to the first data item in the first DATA statement; in this case, the pointer is set to the string constant "Sally" in line 110.

Lines 130 through 170 read two logical records from the DATA statement and print the records on the GS display. The operation is executed as follows. The first time line 140 is executed, the string constant "Sally" is assigned to the variable A$ and the DATA statement pointer moves to the next data item (5). The 5 is assigned to the variable B and the DATA statement pointer moves to the string constant "1305 S.W. Henry St.". This string constant is assigned to the variable C$. Because the end of this DATA statement is reached here, the DATA statement pointer automatically moves to the next DATA statement and points to the first data item in that statement; in this case, the pointer moves to numeric constant 6.95. This value is assigned to the variable D and the DATA statement pointer moves to the string constant "Billy." Since the variable D is the last variable specified in the READ statement, the read operation is finished, and the DATA statement pointer remains pointing to "Billy"; this, of course, indicates that "Billy" is the next data item to be read.

When lines 15Ø and 16Ø are executed, the data items assigned to the variables A$,B,C$, and D are printed on the GS display according to the print format specified in lines 18Ø and 19Ø. (Refer to the IMAGE statement in this section for details on print formats.)

Line 17Ø returns program control to line 14Ø, the READ statement, and the read operation is re-executed. This time the DATA statement pointer is pointing to the string constant "Billy", so "Billy" is assigned to A$. (This, of course, overwrites the previous value of A$ which was "Sally".) The numeric constant 6 is assigned to the variable B, the string constant "15Ø1 S.E. Morrison" is assigned to the variable C$, the numeric constant 5.87 is assigned to the variable D. This ends the second READ operation and the DATA statement pointer is left pointing out into space. This happens because the numeric constant 5.87 is the last data item in the last DATA statement. At this point, the DATA statement pointer must be reset with a RESTORE statement or an INIT statement before another READ operation is executed. If not, a read error occurs and program execution is aborted. In this program, however, a RESTORE statement is not necessary, because the data items just read are printed in lines 15Ø and 16Ø and program execution is terminated in line 2ØØ. The results are shown below:

GS Display Output



Primary Address 34 Specifies the DATA Statement as the Input Source

The DATA statement is specified as the input source for an I/O operation by specifying primary address 34. Normally, primary address 34 is selected by default for READ operations, but if the primary address is specified as a variable in a READ statement, then the variable must be assigned the value 34 to select the DATA statement as the input source. For example:

```
265 READ @A:X,Y,Z$
```

When line 265 is executed under program control, the numeric constant assigned to A specifies the input source. If A equals 34, then the DATA statement is selected as the input source. If A changes to 33, then the current file on the internal magnetic tape unit is selected as the input source. And, if A changes to 15, for example, device number 15 on the General Purpose Interface Bus (GPIB) is selected as the input source.

THE FIND STATEMENT

Syntax Form:

[Line number] FIN [I/O address] numeric expression

Descriptive Form:

[Line number] FIND [I/O address] tape file number

Purpose

The FIND statement positions the magnetic tape head on a peripheral device to the beginning of the specified file. If an I/O address is not specified in a FIND statement, the Graphic System internal magnetic tape unit is selected as the peripheral device by default.

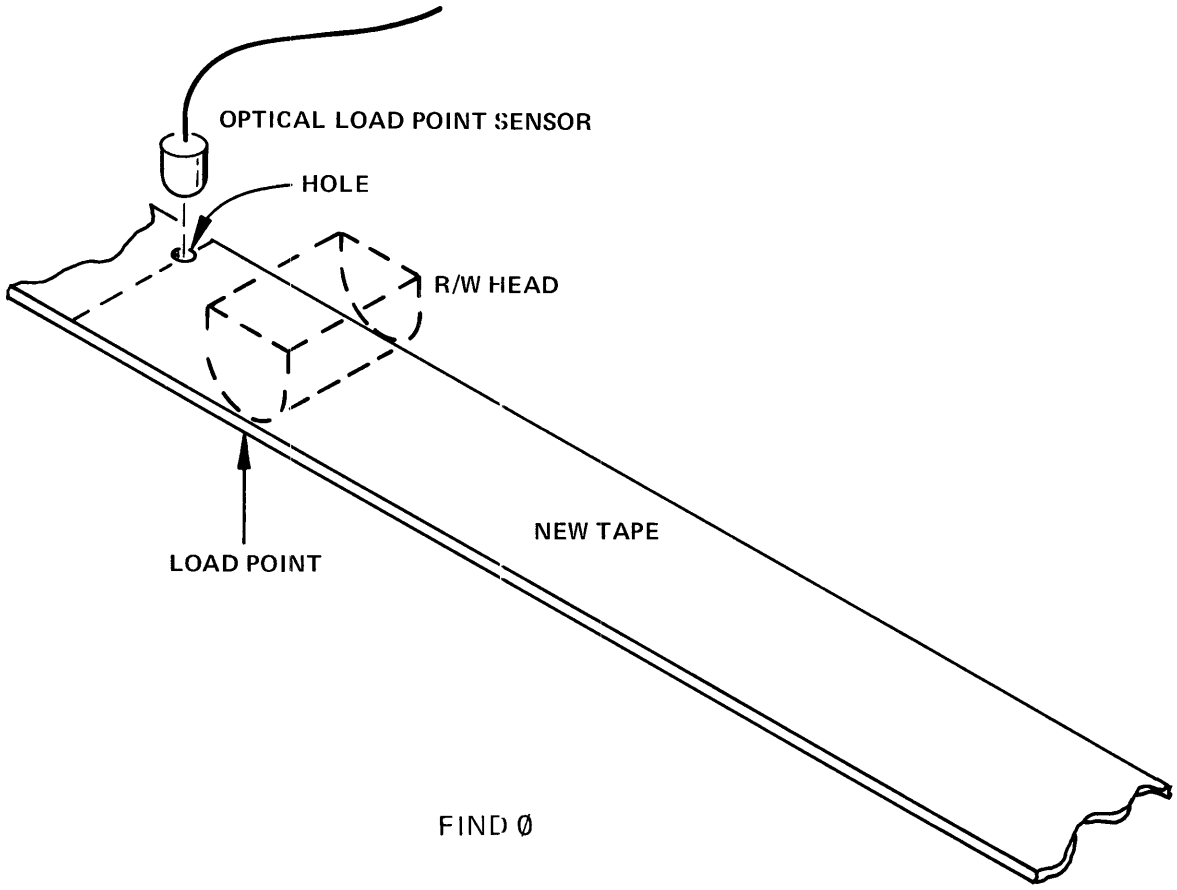
Explanation

Tape File Numbers

Each file on the internal magnetic tape is referenced by a tape file number. The first file on the tape is number 1, the second file is number 2, and so on up to 256. File number Ø refers to the load point at the beginning of the tape. The load point is positioned approximately one inch before the beginning of the first file.

Finding the Beginning of the Tape

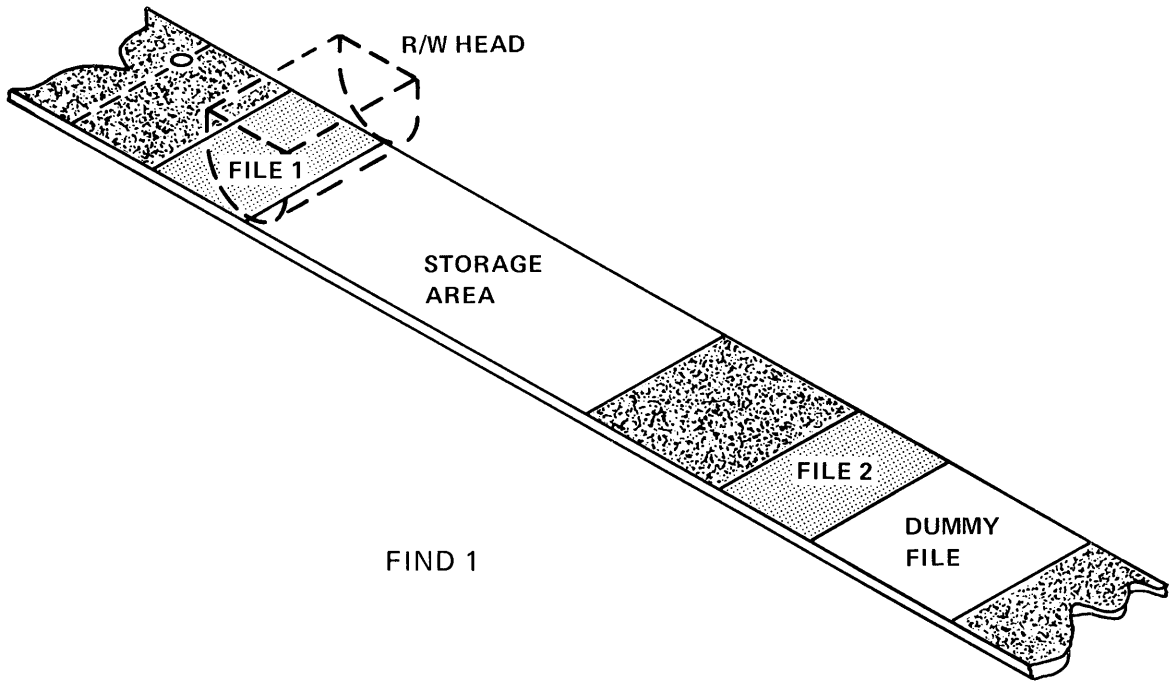
The magnetic tape read/write head is positioned to the load point (the beginning of the tape) by entering the statement FIND Ø and pressing the RETURN key or by executing a FIND Ø statement under program control. (The same results can also be obtained by pressing the REWIND key on the GS keyboard.) The following illustration shows the magnetic tape read/write head positioned at the load point after a FIND Ø statement is executed.



FIND Ø

Finding a Tape File

The magnetic tape head is positioned to the beginning of a tape file by specifying the appropriate file number after the keyword FIND. This method is used to find new (empty) files, ASCII program files, ASCII data files, binary program files, and binary data files. The tape head is always positioned to the beginning of the storage area which is located just past the file header. (This is true unless the magnetic tape status parameter is set to "no header" format. Refer to Magnetic Tape Status in the Environmental Control section for details.) The following illustration shows the position of the tape head after a FIND 1 statement is executed. (This assumes the magnetic tape status parameters are set to their normal values.) File number 1 on the tape must be created with the MARK statement before this FIND 1 statement is executed.



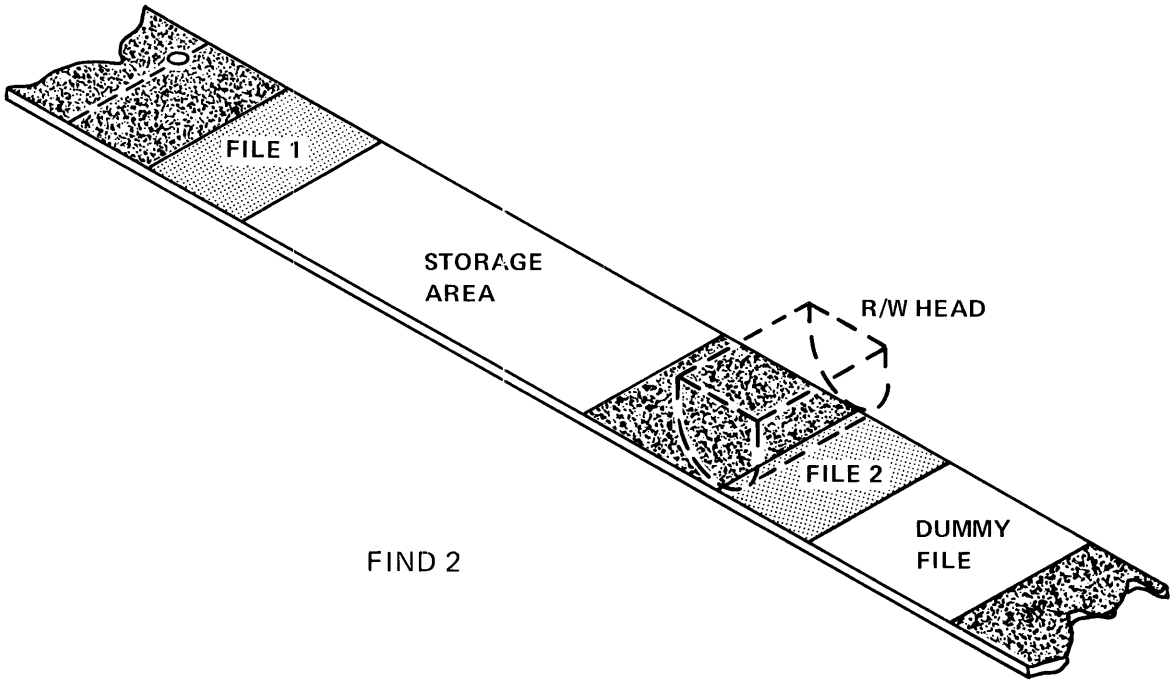
FIND 1

After the specified file is found, information in memory can be stored in the file if the file is new (just created with the MARK statement), or if the file is an old file just killed with the KILL statement. The BASIC program currently in memory is transferred to the file with the SAVE or CALL "BSAVE" statement or data is transferred to the file in ASCII format with the PRINT statement or binary format with the WRITE statement. (Refer to each of these keywords in this section for details.)

If the file already contains information, then the FIND statement opens the file for access. This is analogous to opening a disc file on a mass storage device. If the file contains a BASIC program, executing an OLD, APPEND, CALL "BOLD", CALL "BAPPEN", or CALL "LINK" statement brings the BASIC program into memory. If the file contains ASCII data, then data items are brought into memory with the INPUT statement. If the file contains binary data, the data is brought into memory with the READ statement. (Refer to each of these keywords in this section for details.)

Finding the Last (Dummy) File

The magnetic tape head is positioned to the last (dummy) file by specifying the appropriate file number. The tape head is automatically positioned to the beginning of the file header in preparation for creating new files on the tape with the MARK statement. If the last file number is not known, a TLIST statement can be executed to find it. When a MARK statement is executed after finding the last file, the dummy file is overwritten with new files as they are created, and a new dummy file is created on tape as the last file. The following illustration shows the position of the tape head when the last file is specified in a FIND statement.



FIND 2

Reading the Tape File Header

If a magnetic tape status parameter is changed, the FIND statement positions the tape head to the beginning of the specified file header instead of to the beginning of the storage area. This allows direct access to the file header. Information in the header can be changed or deleted, or new information can be added. For example:

```
100 INIT
110 PRINT @33,0:0,0,1
120 FIND 1
130 INPUT @33:A$
140 PRINT A$
```

When this program is executed, the file header for file number 1 is input into memory and printed on the GS display. Line 1ØØ initializes the system. Line 11Ø changes the internal magnetic tape status parameters to "no header" format. This causes the magnetic tape to position the read/write head to the beginning of the header on file number 1 (line 12Ø). (The magnetic tape unit assumes that the file header is the first logical record in the storage area because a "no header format" is specified.) In line 13Ø, the file header is input into memory and assigned to the string variable A$. In line 14Ø, the header information is printed on the GS display for viewing.

Changing a Tape File Header

A tape file header can be changed as long as numeric digits are not added to the header and as long as the header meets the following minimum format requirements:

1. Character positions 2 through 4 must be a decimal number from 1 to 256. This number represents the file number.
2. Character position 9 must be an A, B, N, or L.

   A = ASCII File
   B = Binary File
   N = New File
   L = Last File
3. Character position 17 must be a P or D.
4. Character position 27 must be an S if a secret program is stored in the file.
5. Character positions 35 through 39 must hold a decimal number from 1 through 65535. This number indicates the number of physical records in the file.
6. Character position 43 must be a Carriage Return (CR) and character position 44 must be a DC3 control character or a blank. The CR/DC3 combination acts as the delimiter to the ASCII character string representing the file header.

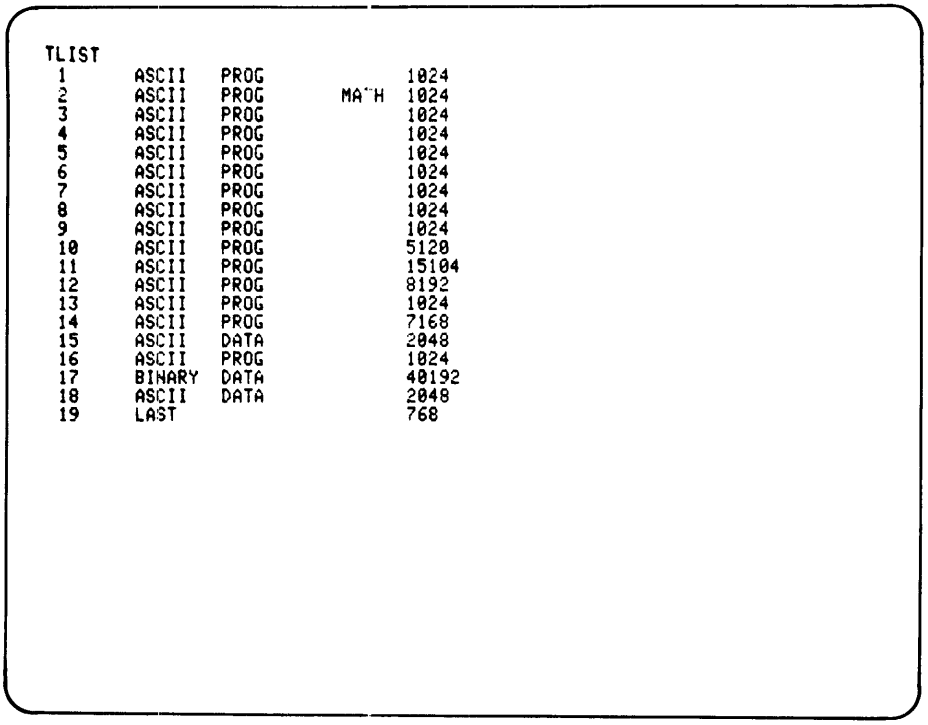
The following program illustrates how to add information to a file header. This program adds the label MATH to the file header where the word SECRET normally resides. This helps identify the content of the program when the file headers are listed in the TLIST statement.

```
1ØØ INIT
11Ø PAGE
12Ø PRINT @33,Ø:Ø,Ø,1
13Ø FIND 2
14Ø INPUT @33:A$
15Ø A$=REP("MATH",28,4)
16Ø FIND 2
17Ø PRINT @33:A$
18Ø PRINT @33:"S"
19Ø PRINT @33,Ø:Ø,Ø,Ø
2ØØ TLIST
```

When line 1ØØ is executed, the system environmental parameters are initialized; line 11Ø clears the display; and line 12Ø sets the internal magnetic tape status to "non-header" format. (Refer to Magnetic Tape Status in the Environmental section for details on setting the internal magnetic tape status parameters.) When line 13Ø is executed, the tape head is positioned to the beginning of the header in file number 2. Line 14Ø then inputs the tape file header information as a character string and assigns the strings to A$.

The best way to change the tape file header is to do a string replace operation as shown in line 15Ø. In line 15Ø, the REP function is used to insert the substring "MATH" into the header string starting at character position 28. Four characters are deleted before the insertion is made. With the header string modified, the beginning of file 2 is found in line 16Ø and the modified header is transferred back to the file in line 17Ø. Line 18Ø prints a DC3 control character CTRL S (S) on the tape right after the CR (Carriage Return) from the last statement and line 19Ø returns the tape status to its normal state. A TLIST operation displays the results of the program on the GS display (shown below) and the program is automatically ended.

GS Display Output



Any characters except digits Ø-9 can be added to the tape file header as long as the information doesn't interfere with the information in columns 2 through 4, column 9, column 27, columns 35 through 39, column 43 and column 44

Finding a File on an External Magnetic Tape Unit

The FIND statement can also be used to find a tape file on an external magnetic tape unit connected to the General Purpose Interface Bus. For example:

```
FIND @17:25
```

When this statement is executed, the I/O address @17,27: is issued over the GPIB. Primary address 17 tells peripheral device number 17 that it has been selected to take part in the upcoming data transfer. Secondary address 27 is issued by default and tells device 17 that the information to be transferred represents the tape file number for a file to be found. The number 25 is then converted to an ASCII character string and sent to device number 17, most significant digit first. Once the ASCII string is received, it is up to device 17 to read the ASCII data string and position its read/write head to the beginning of the specified file.

THE IMAGE STATEMENT

Syntax Form:

[Line number] IMA any characters except CR

Descriptive Form:

[Line number] IMAGE format string for the print using statement

Purpose

The IMAGE statement specifies the print format to be used in a PRINT USING statement. The print format is specified as a "format string."

Explanation

Format String Defined

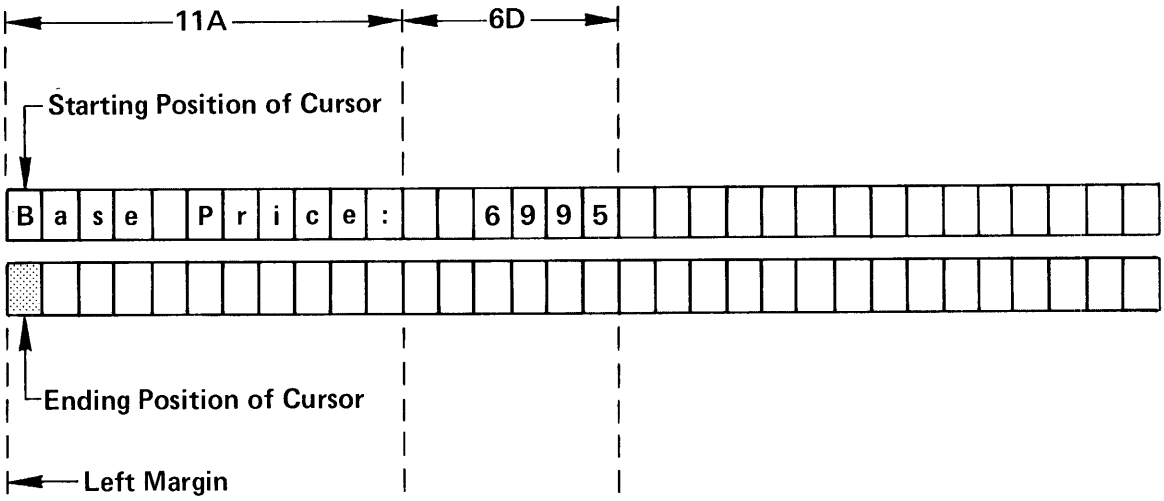
A format string is a special group of characters which guide the BASIC interpreter when it outputs ASCII data via the PRINT USING form of the PRINT statement. Each character in the format string has special meaning. For example, the letter A means "an alphanumeric character must be printed here," the letter D means "a numeric value must be printed here," and the letter X means "add a space character to the ASCII data string at this point." The order in which data items are specified in the PRINT USING statement must closely match the format string which is used as a guide. For example:

```
100 PRINT USING 110: "Base Price:", 6995
110 IMAGE 11A,6D
```

In this example, the print format is specified in the IMAGE statement in line 110. The format string 11A,6D defines two "print fields" or "print zones." The first print field (11A) specifies that the first data item in the PRINT USING statement must be a character string of not more than eleven characters in length. The second print field (6D) means that the second data item in the PRINT USING statement must be a numeric constant (or numeric expression) of not more than six digits to the left of the decimal point. (The decimal part of the number, if any, is rounded off in this case.)

When line 1ØØ is executed, the alphanumeric string "Base Price:" and the numeric constant "6995" are sent to a specified peripheral device in the format specified in line 11Ø, the IMAGE statement. The default print format normally used for ASCII output is suppressed. In this case, an I/O address is not specified, so the ASCII information is sent to the GS display by default. The results are shown below:

GS Display Output



Notice that each print field specified in the format string has an accompanying data item specified in the PRINT USING statement. The data items are also of the correct type; a character string for the A field and a numeric value for the D field. If the data items were reversed in line 1ØØ, a data mismatch would occur, and program execution would abort. If one of the data items did not fit into the field (i.e., if the character string were 12 characters instead of 11 characters), then a field overflow error would occur, and program execution would abort. And, if too many data items were specified in line 1ØØ (three data items, for example instead of two), then a fatal error would occur, and program execution would again abort. So, it is important that the type of data items specified in a PRINT USING statement, and the order in which they are specified, closely match the specifications of the format string.

Field Operators

Field operators are special characters in the format string that define a print field or a special function. For example, the field operator "A" defines an alphanumeric print field for character strings and the field operator "D" defines a numeric print field for numeric values. Only character strings can be printed in A fields and numeric data in D fields. The table below summarizes the purpose of each field operator.

FIELD OPERATOR	NAME	DESCRIPTION
A	Character String	Defines a print field for alpha-numeric character strings. Specified in the form nA where n represents an integer from 1 through 255.
D	Numeric	Defines a print field for numeric data written in standard notation. Specified in the form nD where n represents an integer from 1 through 255.
E	Scientific Notation	Defines a print field for numeric data written in scientific notation. Specified in the form nE where n represents an integer from 1 through 11.
L	Line Feed	Specifies the insertion of a Line Feed character (CTRL J) into the ASCII data string at the specified point. Specified in the form nL where n represents an integer from 1 through 255.
P	PAGE	Specifies the insertion of a PAGE command (CTRL L) into the ASCII data string at a specified point. Specified in the form nP where n represents an integer from 1 through 255.

FIELD OPERATOR	NAME	DESCRIPTION
S	Suppress CR	Specifies the suppression of the Carriage Return character at the end of the ASCII data string. This operator can only appear at the end of the format string.
T	Tab	Specifies a move to a character position in the ASCII data string. Enough spaces are added to the string so that the next printed character appears in the specified column. Specified in the form nT where n represents an integer from 1 through 255.
X	Space	Specifies that space characters be inserted into the ASCII data string at the specified point. Specified in the form nX where n represents an integer from 1 through 255.
"	Literal String	Defines an alphanumeric string to be inserted into the ASCII data string. Specified in the form n" " where n represents an integer from 1 through 255. The characters inside the quotes are "literally" placed in to the ASCII data string.
/	Carriage Return	Specifies the insertion of a Carriage Return character at the specified point in the ASCII data string. Specified in the form n/ where n represents an integer from 1 through 255.

FIELD OPERATOR	NAME	DESCRIPTION
(	Begin Repeat	Specifies the beginning point for repeat instructions on field format. Specified in the form n(where n represents an integer from 1 through 255.
)	End Repeat	Specifies the ending point for repeat instructions on field format.
,	No Operation	Although not required, commas may be inserted between field operators in a format string to increase readability in a program listing. Commas in the format string have no effect on the final output format.

Field Modifiers

Field modifiers are special symbols used in combination with field operators to define the length of the field and to enhance the field.

For example, the n field modifier specifies the number of character positions in the print field. n must be an integer from 1 through 255. Another example of a modifier is the dollar sign field modifier ($). This modifier specifies that a dollar sign be placed in front of the numeric value in a D field.

The following table describes the purpose of each field modifier.

FIELD MODIFIER	PURPOSE
n	Specifies the number of character positions in a print field or specifies the number of times a field operator is repeated. For example, 3D specifies a three position numeric field, and 10X specifies the insertion of 1Ø space characters into the ASCII data string. n must be an integer from 1 through 255, except when used with the E field operator. When used with the E field operator, n must be an integer from 1 through 11.
F	Specifies a print field large enough to accommodate the data item associated with the field. For example, FD creates a 1 digit numeric field if the associated data item has 1 digit, a 6 digit field if the associated data item has 6 digits, and a 1Ø digit field if the associated data item has 1Ø digits.
+	Specifies that a plus sign (+) be placed in front of the numeric value in the print field if the number is positive, and a minus sign (−) if the number is negative. Used with D field operators only.
−	Specifies that a space be placed in front of a numeric value if the number is positive, and a minus sign if the number is negative. Used with D field operators only.
.	Specifies that a decimal point character be placed at a specified location in the ASCII data string. This modifier "links" the D field operator for the integer part of a number to the D field operator that specifies the decimal part of the number. Used with D field operators only.
$	Specifies that a dollar sign ($) be placed in front of the numeric value in the print field. If a plus or minus field modifier is used with the D operator, then the dollar sign is placed to the left of the plus or minus sign. Used with D field operators only.
C	Specifies that commas be inserted into a numeric print field to the left of the decimal point to break the integer part into thousands, millions, etc. Each comma takes up one character position in the field. Used with D field operators only.

Format String Length

The illustrations used in the following examples represent the ASCII data string after it is converted to conform to the print format specified in the associated format string. The first character on the left of each illustration represents the first character sent to the specified peripheral device. This character occupies "position 1." The character positions to the right are "position 2," "position 3," and so on The examples are limited to 32 character positions for ease of illustration. Remember, however, that the line actually extends to an almost infinite number of character positions.

Creating Format Strings

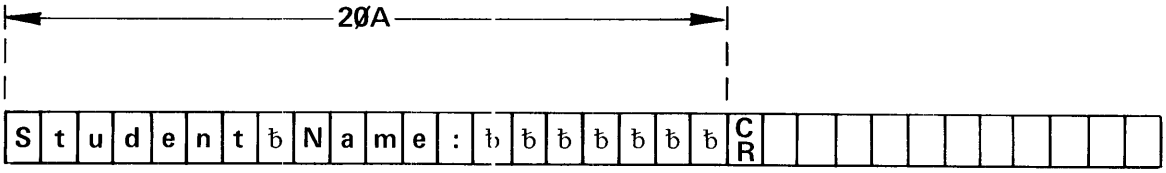
The following examples illustrate how to combine field operators and field modifiers into format strings. All combinations are not covered; however, enough examples are provided to give an understanding of the rules governing format string construction. Most of the programs in these examples send the ASCII data string to the GS display because an I/O address is not specified in the PRINT USING statement. However, by specifying the appropriate primary address in the PRINT USING statement, the ASCII data string can easily be sent to the internal magnetic tape unit or to an external peripheral device over the General Purpose Interface Bus.

The Character String Field Operator (A)

The A field operator defines a print field for alphanumeric character strings. The operator is specified in the form nA where n represents an integer from 1 through 255. If n isn't specified, then 1 is assumed to be the value of n by default.

Example 1—Creating an Alphanumeric Print Field

```
120 IMAGE 20A
130 PRINT USING 120 "Student Name:"
```



ASCII Data String

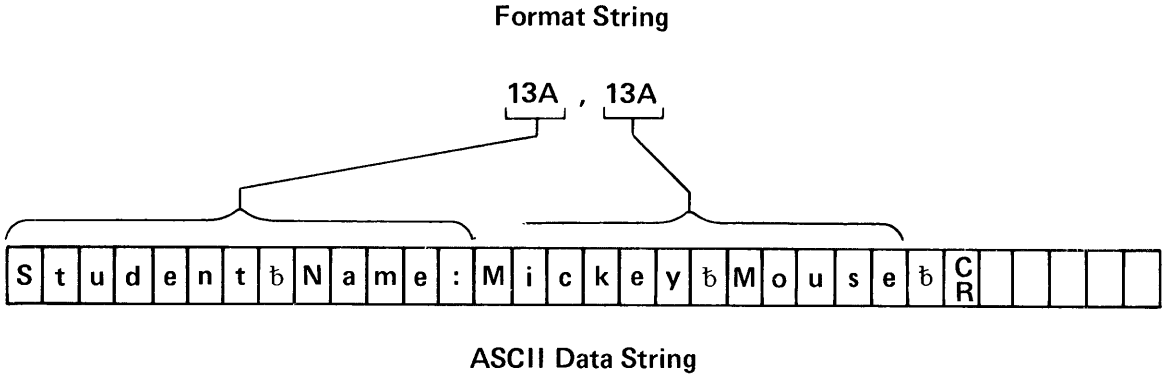
In line 120 above, the format string 20A defines an alphanumeric print field with 20 character positions. In line 130, the string constant "Student Name:" is printed in this field. Notice that the character string is left justified in the field; this means the first character in the string is printed

in the left-most position in the field, the next character in the position immediately to the right, and so on. The character string "Student Name:" only fills 13 character positions; the remaining positions are filled with space characters (represented by the ƀ symbol). Character strings smaller than the specified field length can be printed in the field, but a character string larger than the specified field length cannot be. If an attempt is made to print a string with 21 characters in this field, for example, then a FIELD OVERFLOW ERROR occurs and program execution is aborted.

Another point of interest in this example is the Carriage Return character added to the end of the ASCII data string. This is always done unless otherwise specified. The Carriage Return character acts as a string delimiter, and if sent to the GS display, it is converted into a Carriage Return/Line Feed combination. This of course returns the display cursor to the left margin and moves the cursor down one line.

Example 2—Two Alphanumeric Fields Side by Side

```
14Ø A$="Student Name:"
15Ø B$="Mickey Mouse"
16Ø IMAGE 13A,13A
17Ø PRINT USING 16Ø:A$,B$
```



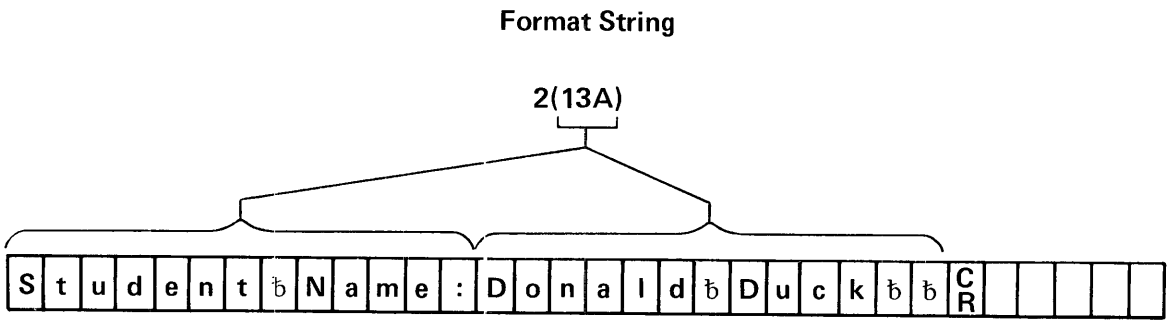
This example brings out the fact that the end of one print field marks the beginning of the next print field. Also, there must be a data item specified in the PRINT USING statement for each print field defined in the format string; A$ for the first field (13A) and B$ for the second field (13A).

The Repeat Field Operators ( )

The repeat field operators specify that the portion of the format string inside the parenthesis is to be repeated n number of times. The operators are specified in the form n( ) where n represents an integer from 1 through 255. If n isn't specified, then 1 is assumed to be the value of n by default. The field operators and modifiers to be repeated are placed inside the parenthesis.

Example 3—Using Repeat Field Operators

```
180 A$="Student Name:"
190 C$="Donald Duck"
200 IMAGE 2(13A)
210 PRINT USING 200:A$,C$
```



This example produces the same results as example 2. The only difference is the addition of repeat field operators in the format string. If a field is repeated two or more times in succession, then using repeat field operators provides a good method for shortening the length of the format string. There must be a data item specified in the PRINT USING statement for every repetition of the field. In this case, a 13 character field is specified twice, so there are two character strings (A$ and C$) specified in the PRINT USING statement (line 210). If the format string were 3(13A), then three character strings would have to be specified in the PRINT USING statement.

The repeat field operators can be nested up to four deep; that is, parenthesis can be placed inside parenthesis up to four deep. For example, 280 IMAGE 2(2(2(2(6A)))).

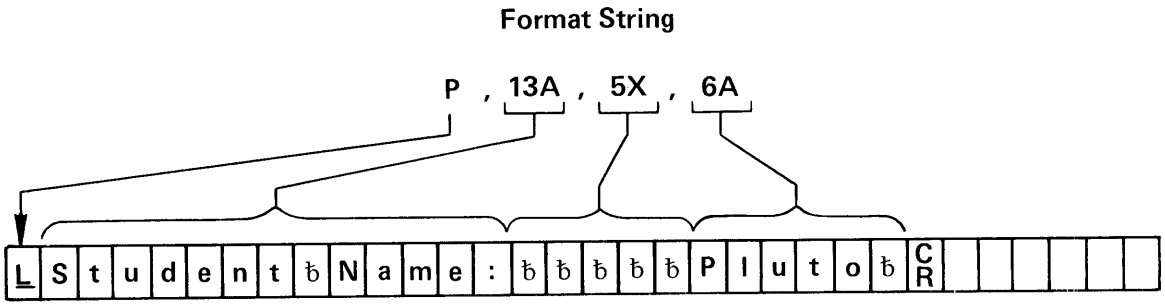
The Space Field Operator (X) and the PAGE Field Operator (P)

The space field operator specifies that one or more space characters be inserted into the ASCII data string at the specified point. The space operator is specified in the form nX where n represents an integer from 1 through 255. If n isn't specified, then 1 is assumed to be the value of n by default.

The PAGE field operator specifies that one or more Form Feed characters be inserted into the ASCII data string at a specified point. The Form Feed character erases the GS display and returns the cursor to the "HOME" position. On an external peripheral device such as a line printer, the Form Feed character usually causes the printer to advance to the next page of paper. The PAGE operator is specified in the form nP where n represents an integer from 1 through 255. If n isn't specified, then 1 is assumed to be the value of n by default.

Example 4—Turning the Page and Adding Spaces between Print Fields

```
220 A$="Student Name:"
230 D$="Pluto"
240 IMAGE P,13A,5X,6A
250 PRINT USING 240:A$,D$
```



In this example, the field operator P is specified first in the format string. This places a Form Feed control character (CTRL L) in the first character position of the ASCII data string. When the ASCII data string is sent to the GS display, a PAGE command is executed before the characters are printed.

This example also shows how to separate two print fields with spaces. The X field operator places 5 space characters between the two A fields as shown in the illustration. Notice that the

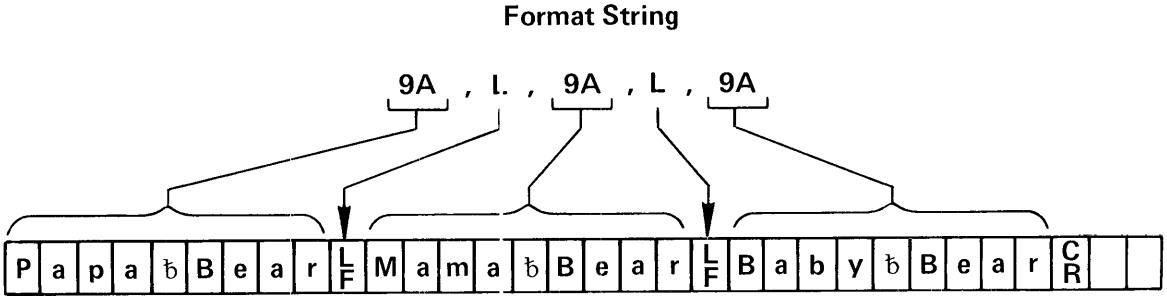
commas in the format string have no effect on the final print format. All they do is increase the readability of the format string in a program listing. The format string can also be specified as P13A5X6A, or P 13A 5X 6A.

The Line Feed Field Operator (L)

The line feed field operator specifies the insertion of one or more line feed characters into the ASCII data string at the specified point. The operator is specified in the form nL where n represents an integer from 1 through 255.

Example 5—Inserting Line Feed Characters into the ASCII Data String

```
260 E$="Papa Bear"
270 F$="Mama Bear"
280 G$="Baby Bear"
290 IMAGE 9A,L,9A,L,9A
300 FIND 0
310 MARK 1,1000
320 FIND 1
330 PRINT @33:USING 290:E$,F$,G$
340 FIND 1
350 INPUT @33:J$
360 PRINT "L";J$
```

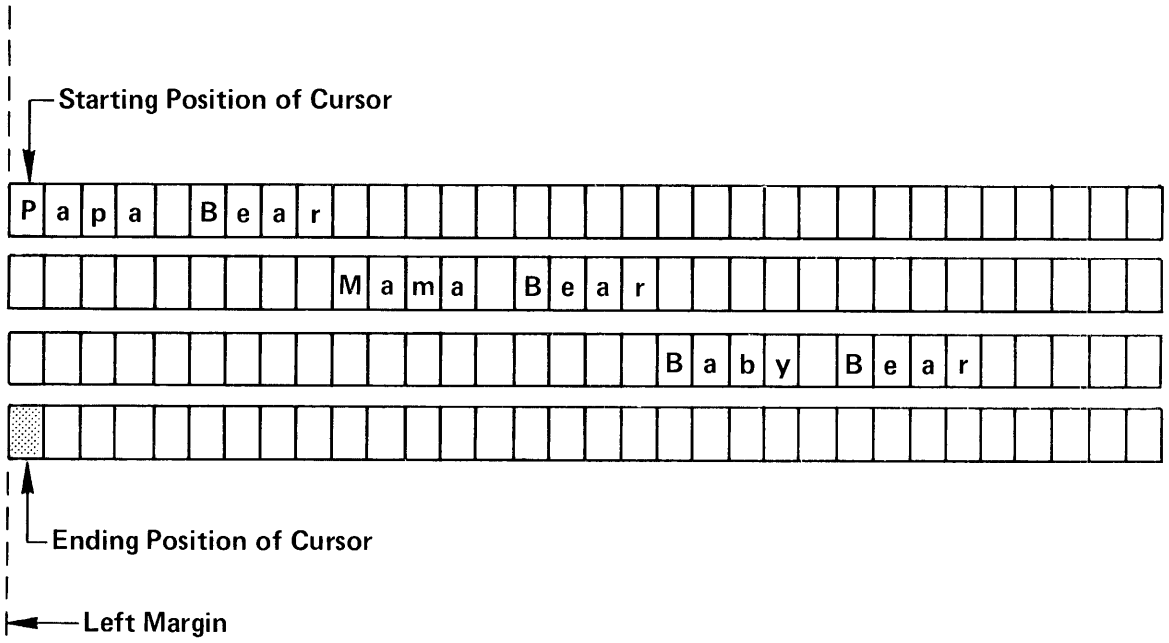


This example illustrates two important formatting techniques. First, how to insert Line Feed characters into the ASCII data string; and second, how to send the formatted ASCII data string to a peripheral device such as the internal magnetic tape unit.

Line Feed characters are inserted into the ASCII data string by specifying the L field operator in the format string. This is done in line 29Ø and the results are shown in format illustration.

The program in example 5 also illustrates how to find the beginning of a new tape (line 3ØØ), create a 1ØØØ byte file (line 31Ø), find the beginning of the new file (line 32Ø), and send three student names to the file for storage (line 33Ø). The program continues by closing the file and positioning the tape head at the beginning of the file (line 34Ø), then inputs the three student names back into memory (line 35Ø) and sends the names to the GS display. The results are shown below:

GS Display Output



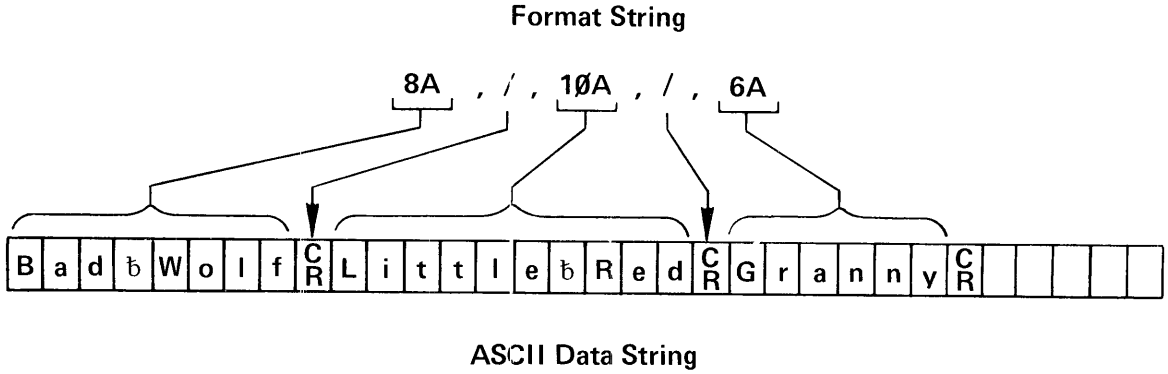
The thing to notice is that the Line Feed characters are indeed inserted into the ASCII data string. The problem with this method of transfer, however, is that the the student names are transferred to the magnetic tape as three separate data items (E$, F$, and G$) and recovered from the tape as one data item (J$). The reason for this is the three data items are transmitted as a single ASCII character string with a Carriage Return on the end. When the information is input back into memory (line 350), all of the characters up to the first Carriage Return are assigned to the first string variable (J$). To get around this problem without having to use three separate PRINT statements, Carriage Return characters can be inserted between each data item when the information is sent to the tape. This leads into the next topic.

The Carriage Return Field Operator (/)

The Carriage Return field operator specifies the insertion of one or more Carriage Return characters into the ASCII data string at the specified point. This operator is specified in the form n/ where n represents an integer from 1 through 255. If n isn't specified, then 1 is assumed to be the value of n by default.

Example 6—Inserting Carriage Returns Between Print Fields

```
370 K$="Bad Wolf"
380 L$="Little Red"
390 M$="Granny"
400 ON EOF(0) THEN 440
410 FIND 1
420 INPUT J$
430 GOTO 420
440 PRINT @33:USING 450:K$,L$,M$
450 IMAGE 8A,/,10A,/,6A
```



This example locates the logical end of the ASCII data file created in the last example, and sends three more student names to the file for storage. This time, however, a Carriage Return character is inserted between each character string so the student names can be recovered individually rather than as a group.

In lines 37Ø, 38Ø, and 39Ø, the three student names are assigned to string variables. In line 4ØØ, an EOF (End Of File) ON unit is activated to alert the BASIC interpreter when the End Of File mark is reached on the tape. Line 41Ø positions the magnetic tape read/write head at the beginning of file 1. Lines 42Ø and 43Ø then input ASCII data items in a repetitive loop and assigns them to J$ until an attempt is made to input the End Of File mark. The only reason these two lines are in the program is to read through the sequential access data file to get to the EOF mark.

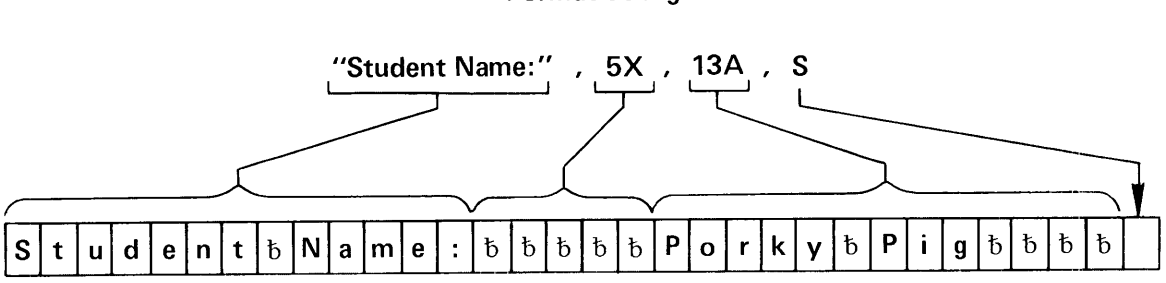
When the EOF mark is found, program control is transferred to line 4ØØ, then to line 44Ø where the three student names are added to the logical end of the file. The ASCII data string is formatted according to the format guide specified in line 45Ø. Notice that the / field operator is used to insert a CR between each data item.

The Literal String Field Operator (") and the S Field Operator

The literal string field operator specifies that an alphanumeric character string is to be inserted into the ASCII data string at the specified point. The literal field operator is specified in the form n" "when n represents an integer from 1 through 255. If n isn't specified, then 1 is assumed to be the value of n by default. The characters inside the quotation marks are "literally" placed into the ASCII data string at the specified point.

Example 7—Inserting a Literal String and Suppressing the Carriage Return

```
460 F$="Porky Pig"
470 PRINT USING 480:F$
480 IMAGE "Student Name:",5X,13A,S
```



In most of the examples up to now the first character string specified has been "Student Name:". In cases like this, it is easier to place the character string in the format string. This way it doesn't have to be specified as a PRINT parameter every time. Example 7 shows how to place a character string into a format string. The character string must be enclosed in quotation marks. The string is called a literal string because the characters are literally transferred from the format string and placed into the ASCII data string as shown in the illustration.

This example also shows how to suppress the Carriage Return character at the end of the ASCII data string. This happens when the S field operator is specified at the end of the format string. It is important to remember that the S field operator can only be placed at the end of the format string—never in the beginning or in the middle.

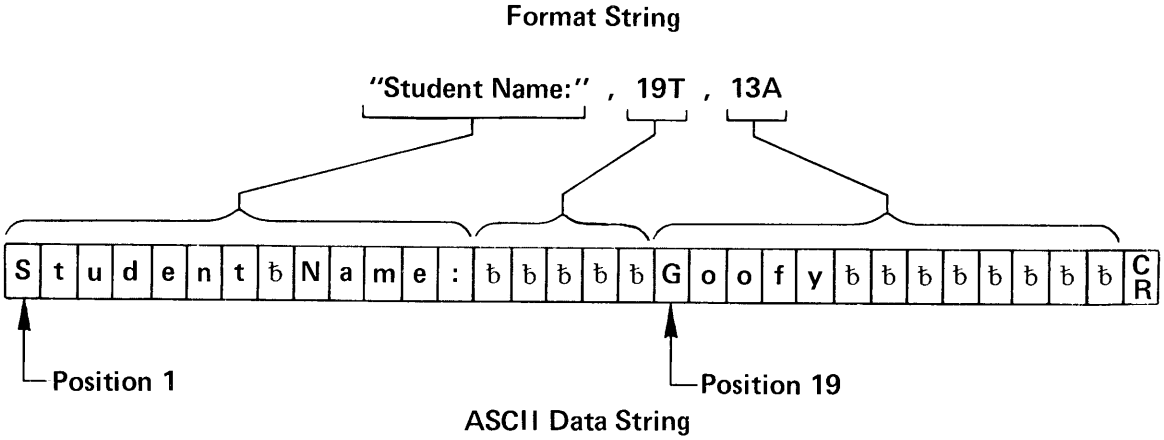
Notice here also that the IMAGE statement does not have to precede the PRINT USING statement in program execution order. The IMAGE statement can be placed anywhere in the program. In addition, several PRINT USING statements can use the same IMAGE statement as a format guide.

The Tab Field Operator (T)

The tab field operator specifies a move to a specific character position in the printed output. For example, if the ASCII data string is printed on the GS display, then the specification 3ØT puts enough spaces in the ASCII data string so that the next printable character is displayed in character position 3Ø. The number of spaces inserted into the ASCII data string depends on the number of Carriage Return characters and the number of control characters which precede the T operator in the format string. The T operator is specified in the form nT where n represents an integer from 1 through 255. If n isn't specified, then a 1 is assumed to be the value of n by default.

Example 8—Tabbing over to a Character Position on the GS Display

```
5ØØ O$="Goofy"
51Ø IMAGE "Student Name:",19T,13A
52Ø PRINT USING 51Ø O$
```



The ASCII data string produced in this example is similar to the ASCII data string in example 7; however, the format string uses the T operator instead of the X operator. In this example, the tab field operator (T) positions the start of the second alphanumeric field to column 19. Notice that enough spaces are added to the ASCII data string so the student name "Goofy" starts in character position 19 on the GS display.

NOTE

The n modifier to T does not specify the number of character positions to the right of the last print field like the n modifier to X. It is important to remember that a tab cannot be executed to a character position to the left of the present position of the display cursor or the writing tool of the external peripheral device. For example, if the format string in line 51Ø is "Student Name:",3T,13A instead of "Student Name:",19T,13A, then a fatal error occurs and program execution is aborted. This happens because the cursor or the writing tool of the external peripheral device is in the 13th character position from the left margin and attempts to cross back over the alphanumeric field to get to the 3rd position from the left margin. This is not allowed.

The Advantage of Using the T Operator Over the X Operator

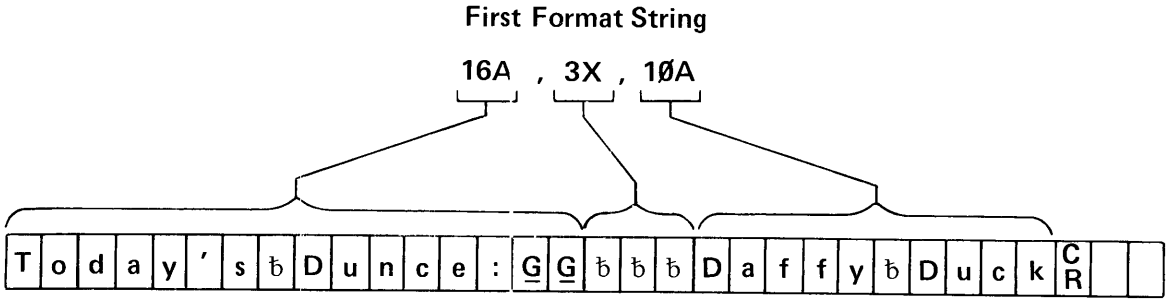
The advantage of using the T field operator over the X field operator is that visual fidelity is always maintained in the printout when the T field operator is used; visual fidelity may not be maintained if the X operator is used. For example, the format strings 16A,3X,1ØA and 16A,19T,1ØA produce the same output (visually) if the character string in the first A field does not contain control characters. If the first field does contain control characters, such as CTRL G (Bell), then the second A field is shifted to the left if the X operator is used instead of the T operator. This happens because the display cursor doesn't move when CTRL G is "printed". (The bell rings instead.) The 19T specification in the second format string makes up for this

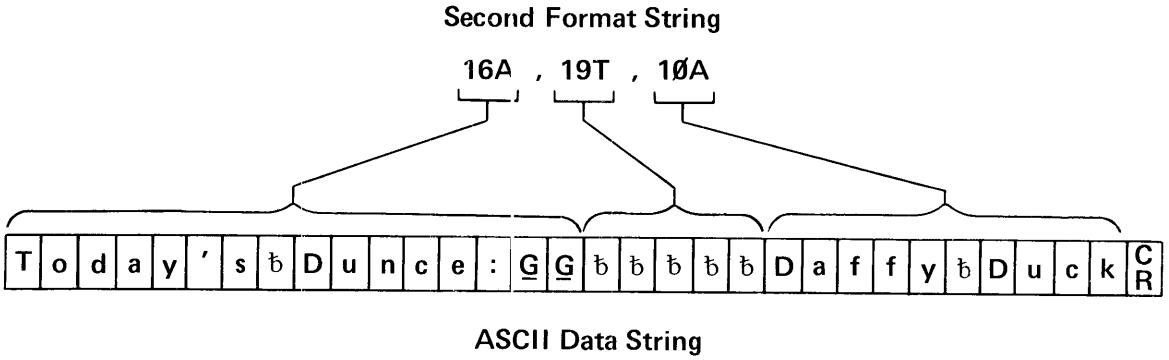
shortcoming. When the tab position is computed by the BASIC interpreter, additional spaces are inserted into the ASCII string to make up for the non-movement of the cursor when control characters are printed. This ensures that the second print field begins at the 19th character position (from the left margin). The following example illustrates how nonprintable control characters in the ASCII data string can make a difference in the appearance of the printout.

Example 9—Visual Fidelity is Maintained When the T Operator is Used

```
530 P$="Today's Dunce:GG"
540 Q$="Daffy Duck"
550 IMAGE 16A,3X,10A
560 IMAGE 16A,19T,10A
570 PRINT USING 550:P$,Q$
580 PRINT USING 560:P$,Q$
```

In this example, the character strings "Today's Dunce:GG" and "Daffy Duck" are printed twice; once using the format specified in line 550 and once using the format specified in line 560. Notice that there are two CTRL G characters in P$. These characters make a difference in the visual appearance of the printout when lines 570 and 580 are executed.

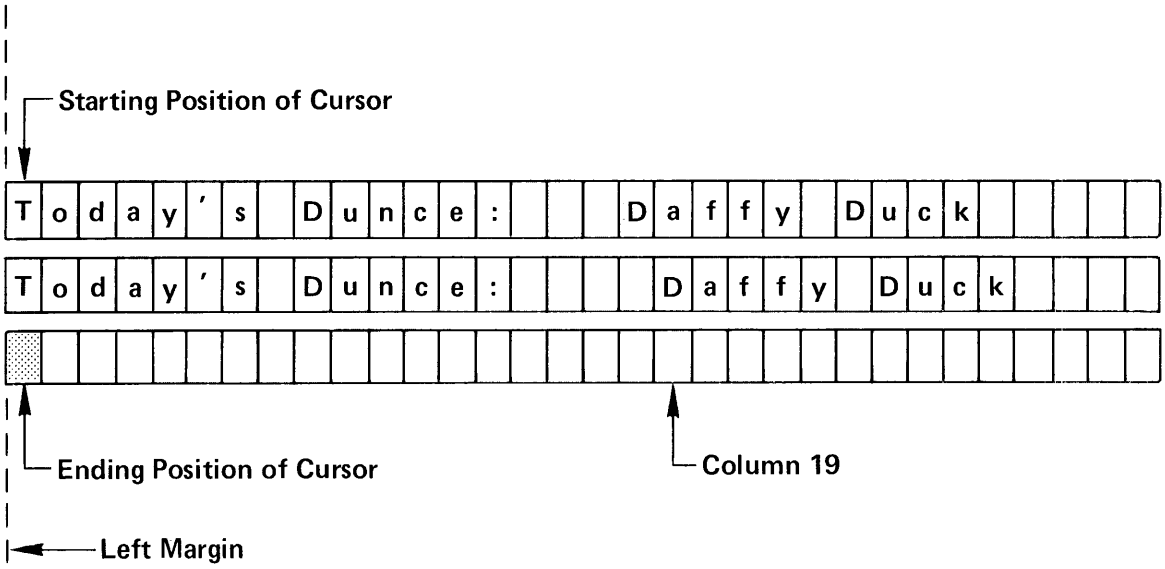




When the ASCII data string is constructed using the format specified in line 55Ø, a character field is created and filled by P$, then three spaces are added as specified by 3X. A 1Ø character field is created next and filled by Q$; after that, a Carriage Return is added onto the end.

When the ASCII data string is constructed again using the format specified in line 56Ø, a 16 character field is created and filled by P$, just as before. This time, however, five spaces are added to the string instead of three. The two control characters in P$ are taken into account when the tab position is computed; the two additional space characters are added to make up for the non-movement of the cursor when the CTRL G characters are "printed". The second A field is created next and filled by Q$; a Carriage Return is added onto the end.

GS Display Output



The printed results on the GS display are shown above. Notice that "Daffy Duck" starts in the 18th character position instead of the 19th position in the first line. This happens because the format string in line 55Ø is used and the control characters in the ASCII data string are not taken into account. In the second line, however, "Daffy Duck" starts in position 19, the specified tab position.

Keeping Track of the Display Cursor Position

The Graphic System has an internal counter which keeps track of the display cursor position. This counter is used as the basis for computing the tab position in a format string. The counter operates under the following rules:

1) The character position counter starts at 1 and increments to 255. A count of 1 means that the display cursor is at the left margin (position 1).

2) Each printable character in the ASCII data string increments the counter by one count as the character is sent out.

3) Control characters (decimal equivalent Ø through 31) in the ASCII data string are not considered printable characters and have no effect on the counter. (Refer the PRINT statement in this section for information on printing control characters.)

4) Every Carriage Return in the ASCII data string resets the counter to 1 (left-hand margin).

5) The counter is reset at the beginning of every new PRINT statement and PRINT USING statement.

The last two rules are of particular importance when specifying the tab position in a format string. Here's why.

Special Case Number 1:

```
59Ø IMAGE 15A,/,2ØT,1ØA
6ØØ PRINT USING 59Ø:A$,B$
```

When these two statements are executed, a 15 position character field is created for A$; a Carriage Return is inserted into the ASCII data string next. If the Carriage Return character were not specified, then the T field operator would add 5 space characters to tab over to position 2Ø (assuming, of course, that there aren't any control characters in the A field). In this case, however, the Carriage Return resets the counter to 1. The BASIC interpreter thinks the cursor is on the left margin and adds 2Ø space characters to the ASCII data string to tab to position 2Ø.

This example emphasizes the point that if a Carriage Return character is in the format string and a T operator follows, then the T operator assumes the display cursor (or the writing tool of the external peripheral device) is located on the left margin and the tab position is computed on that basis.

Special Case Number 2:

```
610 IMAGE 20A,S
620 IMAGE 25T,20A
630 PRINT USING 610:A$
640 PRINT USING 620:B$
```

When line 630 is executed in the program above, the character string assigned to A$ is printed according to the format specified in line 610. A 20 character field is printed on the display and the cursor remains in position 21 because the Carriage Return is suppressed. When line 640 is executed (using line 620 as a format guide) the tab to position 25 is computed. The display cursor is sitting in position 21; however, the character counter is reset to 1 because a new PRINT statement is being executed. As a result, the BASIC interpreter thinks the cursor is in the number 1 position and adds 25 spaces to ASCII data string. As a result, the cursor moves to position 45 instead of position 25 and B$ is printed.

This example emphasizes the point that the character position counter is reset at the beginning of each new PRINT statement and this fact must be considered when specifying the tab position. In this case, the tab specification in line 620 should be 5T to tab to position 25 or the program should be rewritten as follows:

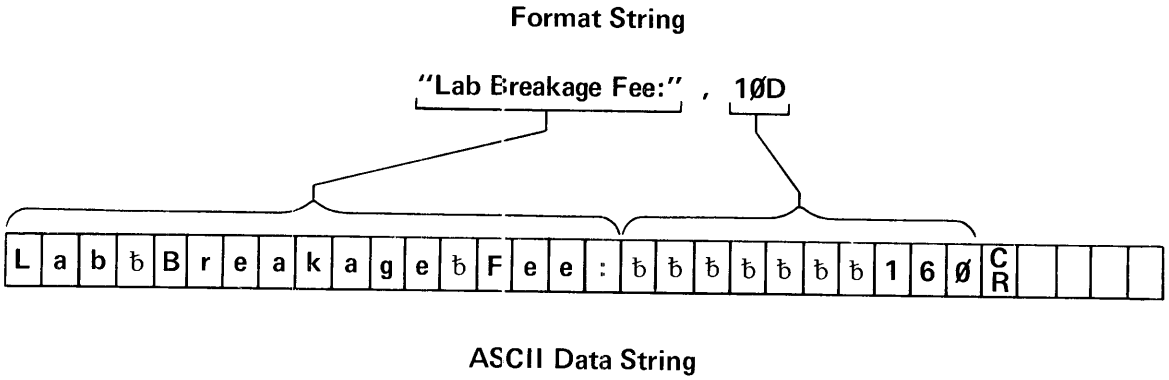
```
650 IMAGE 20A,25T,20A
660 PRINT USING 650:A$,B$
```

The Numeric Field Operator (D)

The D field operator defines a print field for numeric data written in standard notation. The operator is specified in the form nD where n represents an integer from 1 through 255. If n isn't specified, then 1 is assumed to be the value of n by default.

Example 10—Print Fields for Integer Values

```
670 J=159.95
680 IMAGE "Lab Breakage Fee:",10D
690 PRINT USING 680:J
```



This example illustrates how to specify a print field for an integer value. In line 67Ø, the numeric value 159.85 is assigned to the variable J. In line 69Ø, the value of J is printed according to the format specified in line 68Ø, the IMAGE satement. The results are shown in the illustration. Notice that the specification 1ØD defines a ten digit print zone for the numeric value assigned to J. Because the print field is restricted to integer values, the value of J is rounded to an integer before it is placed in the field; in this case, 159.95 is rounded to 16Ø. Notice that the integer is right justified in the field and unused character positions are filled with blanks.

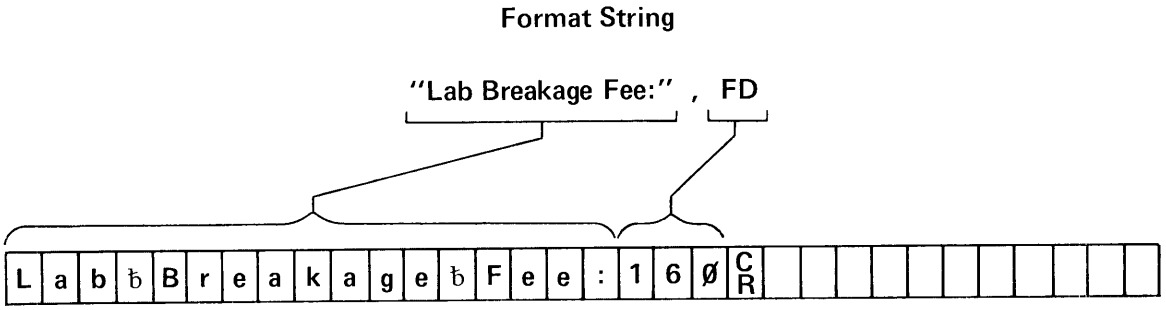
If the integer value is larger than the defined print zone, an error occurs and program execution is aborted. In this case, if the assigned value of J increases to more than 1Ø digits to the left of the decimal point (or 9 digits for negative values), then an error occurs.

Using the F Modifier With a D Field Operator

If an F modifier is used with a D field operator, then a print zone is created just large enough to accommodate the specified data item.

Example 11—Using an F Modifier with a D Field Operator

```
7ØØ A=159.95
71Ø PRINT USING 72Ø:A
72Ø IMAGE "Lab Breakage Fee:",FD
```



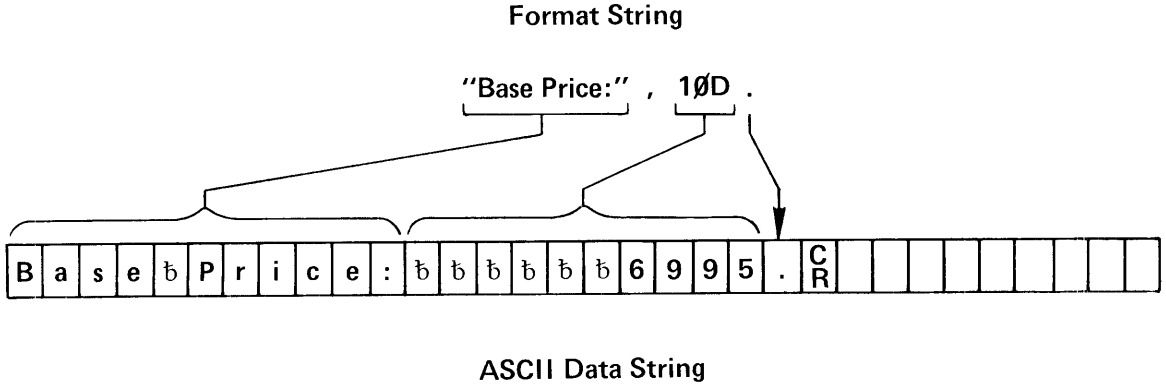
In line 71Ø, the value of B is printed on the GS display according to the format specified in line 72Ø. Notice this time that an FD print field is specified. Because the value of B has three digits, a numeric print field is created with three character positions; just enough to accommodate the number. If the assigned value of B changes to 12345, for example, then a print field with five character positions is created.

The Decimal Point Field Operator (.)

The decimal point field operator defines the location of the decimal point for numeric output. The decimal point operator must follow a D operator with no space or comma in between.

Example 12—Specifying the Location of the Decimal Point

```
73Ø B=Ø6994.95
74Ø IMAGE "Base Price:",1ØD.
75Ø PRINT USING 74Ø:B
```

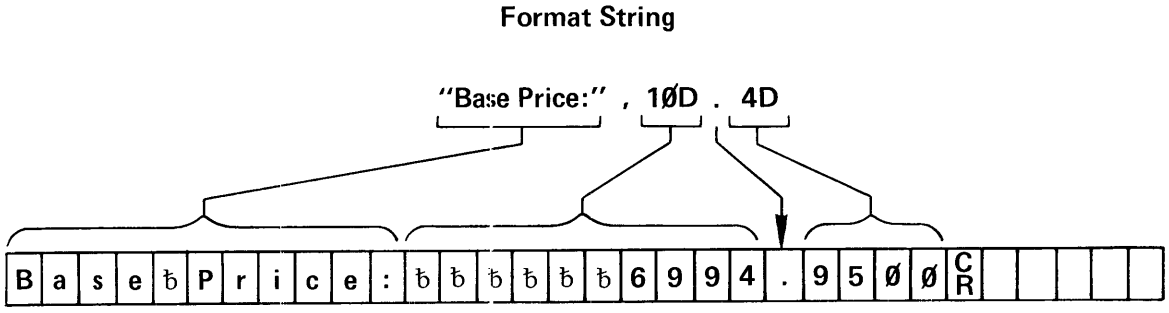


In line 73Ø, the numeric value Ø6994.95 is assigned to the numeric variable B. In line 75Ø, the value assigned to B is printed according to format specified in line 74Ø. This format string includes the decimal point field operator, and the results are shown in the illustration. In this case, only an integer field is specified, so the value of B is rounded to the nearest integer. Notice also that the leading zero is suppressed and not placed in the ASCII data string.

Specifying an Integer Print Field and a Decimal Print Field

Example 13—Specifying a Decimal Print Field with an Integer Print Field

```
76Ø B=6994.95
77Ø IMAGE "Base Price:",1ØD.4D
78Ø PRINT USING 77Ø:B
```



The ASCII data string is formatted the same as example 12 except for the addition of the decimal field specified by 4D. In this case, a four digit numeric field is created to the right of the decimal point. Notice that if the numeric value does not have enough digits to fill the defined decimal field, the empty positions are filled with zeros. In addition, if the numeric value has a decimal part greater than the number of positions defined in the decimal print field, then the decimal part is rounded off to fit the field.

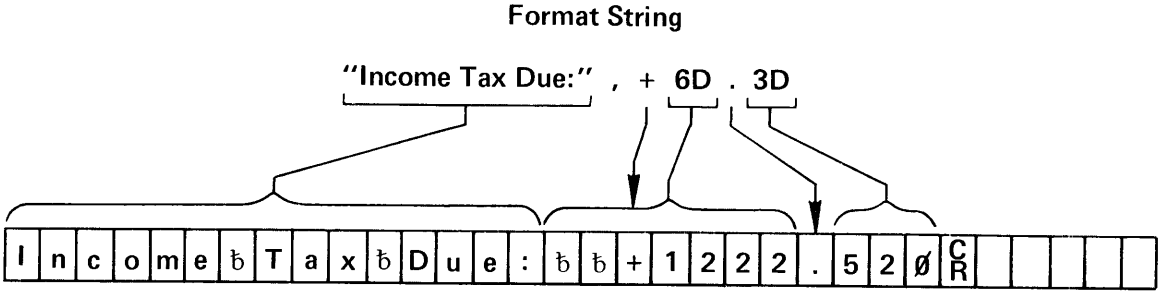
If the F modifier is used with the D field operator, just enough positions are created in the field to accommodate the decimal portion of the number. In this case, the format string "Base Price:",1ØD.FD creates a two digit decimal field because 6994.95 has a two digit decimal part.

Using the Plus Sign Modifier (+) with the D Field Operator

The plus sign (+) modifier causes the BASIC interpreter to place a plus sign in front of a numeric value, if the value is positive, and a minus sign (−) in front of a value, if the value is negative. Another character position is generated in the integer field to accommodate the sign.

Example 14—Using the Plus Sign Modifier

```
790 K=-1222.52
800 PRINT USING 810:K
810 IMAGE "Income Tax Due:",+6D.3D
```



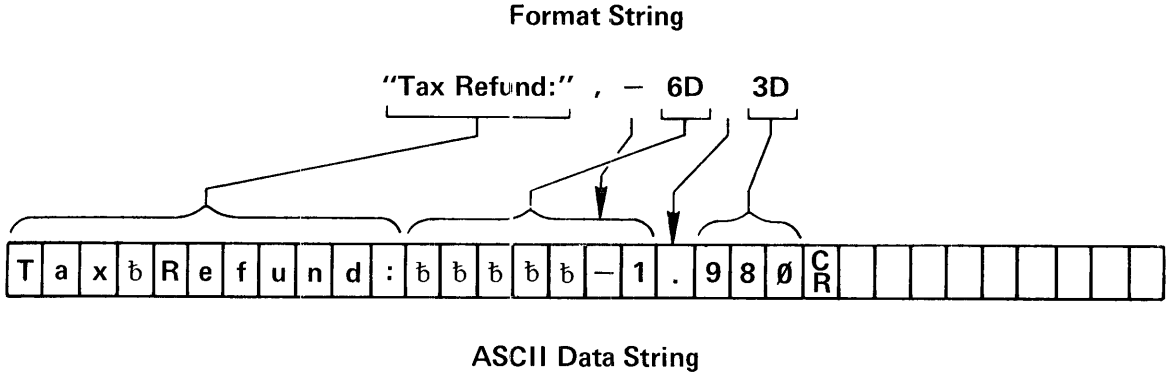
Line 800 in this example outputs the value of K using the format specified in line 810. Because the + modifier is specified in the format string, a plus sign is placed in the integer field to the left of the most significant digit in the field. If the value of K is changed to a negative number, then a minus sign is placed in the field instead of a plus sign. It is important to note here that an additional character position is added to accommodate the plus sign in the integer field.

Using the Minus Sign Modifier with the D Field Operator

The minus sign modifier (−) causes the BASIC interpreter to place a blank (or space) in front of a numeric value, if the value is positive; and a minus sign in front of the value, if the value is negative.

Example 15—Using the Minus Sign Modifier

```
820 K= -1.98
830 PRINT USING 840:K
840 IMAGE "Tax Refund:",-6D.3D
```



This example shows how to specify the minus sign modifier in an integer print field. The only difference between the minus signal modifier and the plus sign modifier is that: the minus sign modifier leaves a blank in front of the most significant digit for positive numbers, and the plus sign modifier, of course, places a plus sign in front of the positive numbers. Notice that in all cases, the plus and minus sign modifiers are placed immediately to the left of the n modifier in the format string (if a C modifier is not specified).

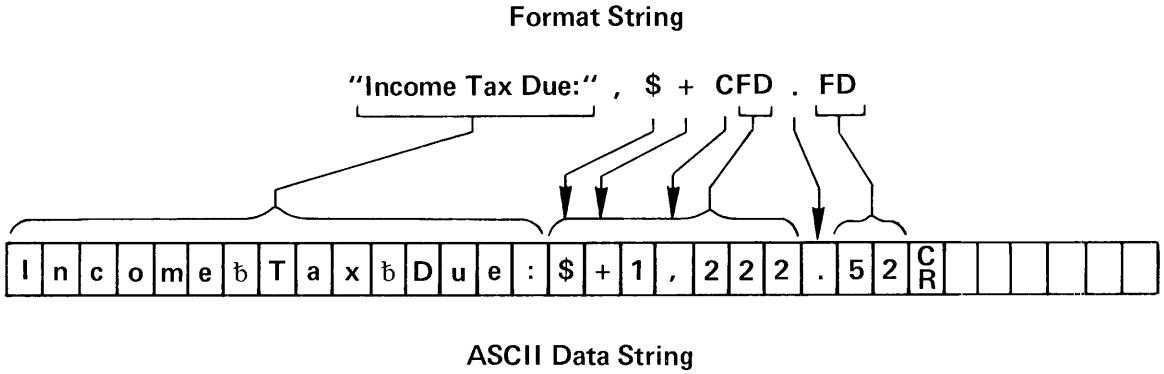
Using a Dollar Sign Modifier ($) and Comma Modifier (C) with the D Field Operator

The dollar sign modifier ($) causes the BASIC interpreter to place a dollar sign in a numeric field to the left of the most significant digit. The dollar sign modifier is always placed to the left of plus sign modifier, a minus sign modifier, or a comma modifier in a format string.

The C modifier causes the BASIC interpreter to place commas in the integer D field to divide the integer part into thousands, millions, etc. Each comma takes up one character position in the D field. Care must be taken when the n modifier is specified to make sure that enough character positions are created to make room for the commas.

Example 16—Using a $ and C Modifier with a D Field Operator

```
850 M=1222.52
860 PRINT USING 870:M
870 IMAGE "Income Tax Due:",$+CFD.FD
```



This example shows how the dollar sign modifier causes the BASIC interpreter to place a dollar sign in front of the integer part of the number. Notice also in this example, that the F modifiers to the D field operators give the most flexibility to the format. Field overflow errors never occur if F modifiers are used and the exponent range of the number is within ±127.

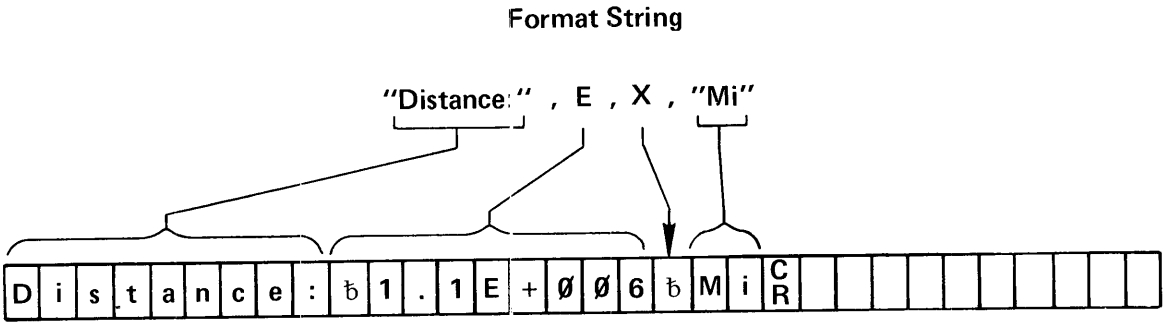
The Scientific Notation Field Operation (E)

The E field operator specifies a print field for numeric output in scientific notation. The E field operator can be modified with the n modifier or F modifier, and the + modifier. The n modifier must be an integer from 1 through 11 and specifies the number of digits to the right of the decimal point in the mantissa. If the F modifier is used in place of the n modifier, up to nine digits are output to the right of the decimal point in the mantissa depending on the number of significant digits in the specified value. For example, if the number to be printed has five digits to the right of the decimal point, then five digits are printed; if the number has seven digits, then seven digits are printed; trailing zeros are suppressed. When the plus sign modifier is used, a plus sign is placed in front of the mantissa, if the mantissa is positive; if the mantissa is negative, a minus sign is placed in front .

There is no control over the exponent format. The exponent is always printed with an E followed by a plus or minus sign followed by three digits. Zeros are added to exponents with less than three digits to give the exponent a uniform output appearance.

Example 17—Specifying an E Field Operator Without a Modifier

```
880 LET R = 1.123456789E+6
890 IMAGE "Distance:",E,X,"Mi"
900 PRINT USING 890:R
```

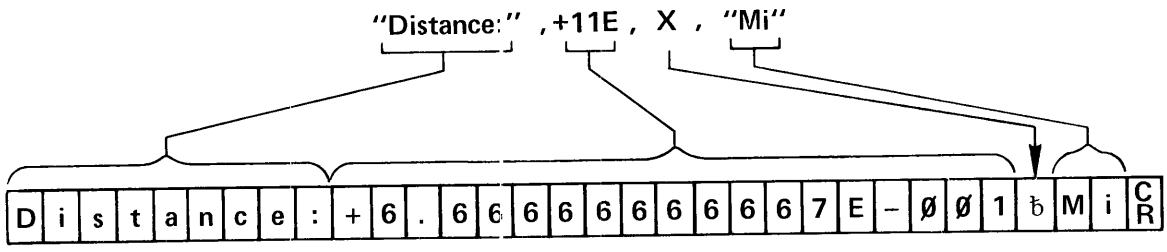


In this example, the field operator E defines the format to be used to print the value of R. Because an n modifier is not specified, a 1 is assumed to be the value of n by default. This is the same as specifying 1E. The E specification directs the BASIC interpreter to round off the decimal part of the mantissa to one digit. The results are shown in the illustration. Notice that only one digit is output to the right of the decimal point. Also notice that two zeros are added to the one digit exponent to give the output a uniform appearance.

In this case, a + modifier is not used, so a space is placed in front of the mantissa. If the mantissa were negative, then a minus sign would have been placed in front of the mantissa.

Example 18—Using a + Modifier and a n Modifier with an E Field Operator

```
910 LET W=2/3
920 IMAGE "Distance:",+11E,X,"Mi"
930 PRINT USING 920:W
```

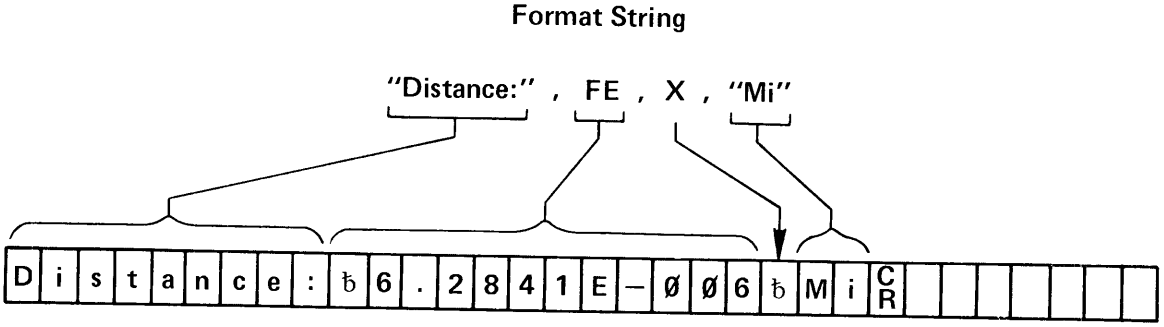


This example shows how to generate the maximum E field length of 19 character positions. The n modifier to E is specified as 11, the largest integer allowed. This causes the BASIC interpreter to print the 11 digits to the right of the decimal point. Notice that the last digit is rounded to the next highest integer.

This example also shows the results of the + modifier in the format string. The + modifier places a plus sign in front of the mantissa for positive numbers and a minus sign in front of the mantissa for negative numbers.

Example 19—Using the F Modifier with the E Field Operator

```
940 G = 6.2841E-6
950 IMAGE "Distance:",FE,X,"Mi"
960 PRINT USING 950: G
```



In this example, the F modifier is used instead of the n modifier. Notice that all of the significant digits to the right of the decimal point in the mantissa are printed. When the F modifier is used, up to nine digits can be output to the right of the decimal point.

Upper Case Letters versus Lower Case Letters in Format Strings

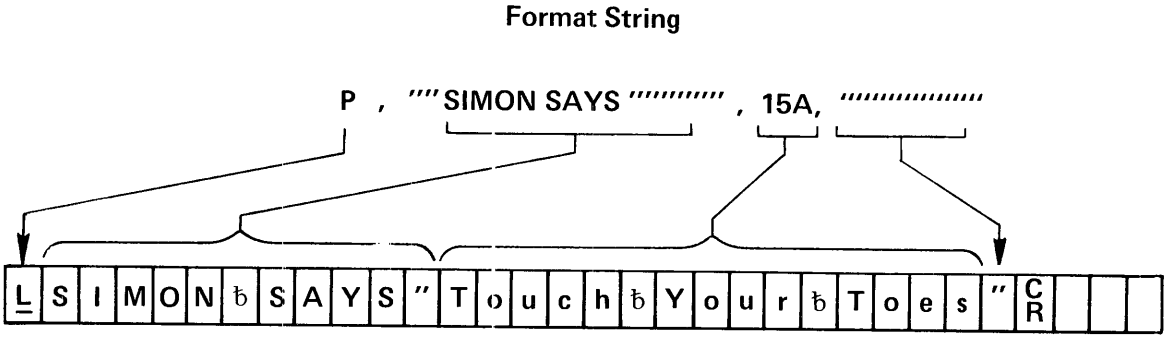
Field Operators and Field modifiers can be specified as either an upper case letter or a lower case letter; it doesn't matter. For example, the format string $+cfd.fd is interpreted the same as $+CFD.FD.

Three Ways to Specify a Format String

There are three ways to specify a format string in a PRINT USING statement. One way is to place the format string in an IMAGE statement and then specify the line number of the IMAGE statement. This method has been used exclusively up to now. Another way is to assign the format string to a string variable like A$, then specify A$ as the format to be used in the PRINT USING statement. A third way is to place the format string directly into the PRINT USING statement. Here are two examples which show how to implement these last two methods.

Example 2Ø—Using a String Variable to Specify a Format String

```
97Ø LET A$="P,""SIMON SAYS""""""""",15A,""""""""""""""
98Ø LET B$="Touch Your Toes"
99Ø PRINT USING A$:B$
```



This example uses a string variable (A$) to specify the format string instead of an IMAGE statement. The format string is assigned to A$ in line 97Ø. Notice that the format string must be enclosed in quotation marks and that all quotation marks within the format string are doubled, including the literal string field operator. In line 99Ø, the character string assigned to B$ is printed using the format specified by A$.

Example 21—Specifying a Format String Directly in the PRINT USING Statement

```
1ØØØ LET B$="Touch Your Toes"
1Ø1Ø PRINT USING "P,""SIMON SAYS""""""""",15A,""""""""""""""":B$
```

This program produces the same results as the program in example 2Ø. The only difference is that the format string is specified directly in the PRINT USING statement. Notice that this method, like the previous method, requires that the format string be enclosed in quotation marks and that any quotation marks inside the format string are doubled. This method of specifying the format string is fine as long as the string is short and simple. If the format string is long and complex, however, this method of specification can make the PRINT USING statement too long to fit on a 72 character line.

THE INPUT STATEMENT

Syntax Form:

[Line number] INP [I/O address] { array variable / string variable / numeric variable } [, { array variable / string variable / numeric variable }] ...

Descriptive Form:

[Line number] INPUT [I/O address] target variables for incoming data items which are formatted in ASCII code

Purpose

The INPUT statement inputs numeric data, array elements, and character strings from the specified input source and assigns the data items to the specified variables. If an input source is not specified, the GS keyboard is selected as the input source by default. The BASIC interpreter assumes the incoming data is formatted in ASCII code.

Explanation

The GS Keyboard

Single Entries. If an INPUT statement is executed without specifying an input source, the GS keyboard is selected as the input source by default. Normally, all keyboard input operations are preceded by a PRINT statement to the GS display which prompts the keyboard operator for input. For example:

```
100 PRINT "What is your Name?"
110 INPUT A$
120 PRINT "This is a Stick-up!"
130 PRINT "How Much Money do You have in Your Wallet?"
140 INPUT B
```

When line 100 in this program is executed, the BASIC interpreter prints the message "What is Your Name?" on the GS display. Line 110 then causes the BASIC interpreter to place a blinking question mark on the screen and wait for an entry from the GS keyboard. Up to 72 characters can be entered into the line buffer before pressing the RETURN key; this includes any letter, number, or special symbol.

When the RETURN key is pressed, the entire contents of the line buffer are assigned to A$. If the RETURN key is pressed without making an entry from the keyboard, then a null string is assigned to A$. It is important to note here that character string assignments made with the INPUT statement do not require enclosing quotation marks. If quotation marks are entered into the line buffer, the BASIC interpreter assumes that they are part of the string and assigns them to the string variable along with the other characters in the line buffer.

In this example, the BASIC interpreter executes lines 12Ø and 13Ø after the RETURN key is pressed and prints the message "This is a stick-up! How much Money do you have in your Wallet?" The BASIC interpreter then waits for additional keyboard input; this time it looks for a numeric value because the numeric variable B is specified in the INPUT statement. When the RETURN key is pressed, the BASIC interpreter evaluates the contents of the line buffer. The first valid numeric entry is assigned to B. All non-numeric entries which precede and/or follow the first numeric entry are ignored. If the RETURN key is pressed without entering a valid numeric constant, the BASIC interpreter keeps displaying the blinking question mark until a valid number is entered and the RETURN key is pressed. In this case, if the keyboard operator enters "$1ØØ.95," the BASIC interpreter assigns 1ØØ.95 to the variable B; the dollar sign is ignored. If the keyboard operator enters "1ØØ dollars and 95 cents," the BASIC interpreter assigns 1ØØ to the variable B and ignores "dollars and 95 cents." And, if the keyboard operator enters "one hundred dollars and ninety five cents," the BASIC interpreter ignores the entire entry and keeps displaying the blinking question mark until a valid number is entered. Numeric constants can be entered in standard format or scientific format. If a previously dimensioned array variable is specified, then the BASIC interpreter keeps requesting data until every array element has an assigned value.

Multiple Entries

If more than one variable is specified in an INPUT statement, an entry must be made for each variable. For example:

```
15Ø INPUT A$,B$,C$,D$,E$
```

When this statement is executed, five entries must be made from the GS keyboard before program execution continues. Each entry is entered into memory by pressing the RETURN key. In this case, all of the characters entered up to the first RETURN are assigned to A$; all of the characters entered up to the second RETURN are assigned to B$, and so on. When E$ is assigned a value, program execution continues to the next statement.

If numeric variables are specified, a valid numeric constant must be entered for each variable. For example:

```
160 INPUT A,B,C,D,E
```

When this line is executed, the BASIC interpreter waits for five numeric entries from the GS keyboard. The entries can be separated by non-valid numeric characters such as commas or spaces, or they can be separated by pressing the RETURN key (or both). If the entries are separated by non-valid numeric characters, the RETURN key must be pressed last to enter the values into memory.

Combinations of string variables and numeric variables can also be specified. In this case, pressing the RETURN key is the only valid delimiting action for string assignments. For example:

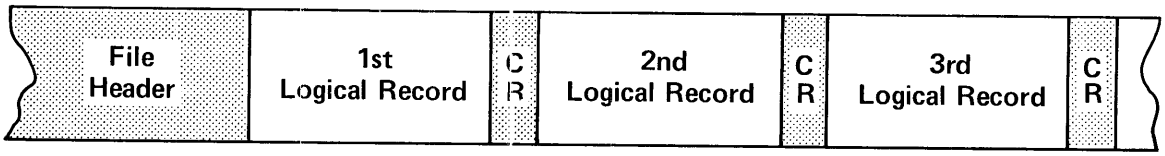
```
170 INPUT A$,B,C$,D,E
```

When this statement is executed, the BASIC interpreter assigns characters to A$ until the RETURN key is pressed. The first valid numeric entry after the RETURN key is pressed is assigned to the variable B, and the remaining characters are assigned to C$ up to the second RETURN. It is important to realize here that if the entries for C$,D, and E are entered into the line buffer and then the RETURN key is pressed, the numeric values for D and E are treated as part of the string and are assigned to C$. So, the C$ entry must be terminated with a RETURN before entering values for D and E. Pressing the RETURN key after the B entry is not necessary however, because the BASIC interpreter terminates the B entry on the first non-numeric character. This character is the first character assigned to C$.

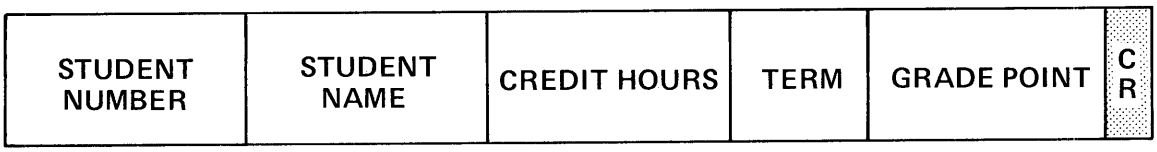
Inputting ASCII Data from the Internal Magnetic Tape

Logical Records

ASCII data files on the internal magnetic tape are subdivided into a series of logical records as shown below:



Carriage Return characters mark the end of each logical record. A logical record can be any given length and can be any combination of numeric data and string data. For example, assume that file number 2 is used to store student records and each record is stored in the following format:



This student record is referred to as a logical record on magnetic tape. The record is composed of five fields. The first field (Student Number) is a numeric value, the second field (Student Name) is a character string, the third field (Credit Hours) is a numeric value, the fourth field (Term) is a character string, and the fifth field (Grade Point) is a numeric value. This logical record is only an example used for illustrative purposes. Logical records of any length and any format can be created with the PRINT statement. (Refer to the PRINT statement in this section for a detailed explanation on creating and storing logical records on magnetic tape.)

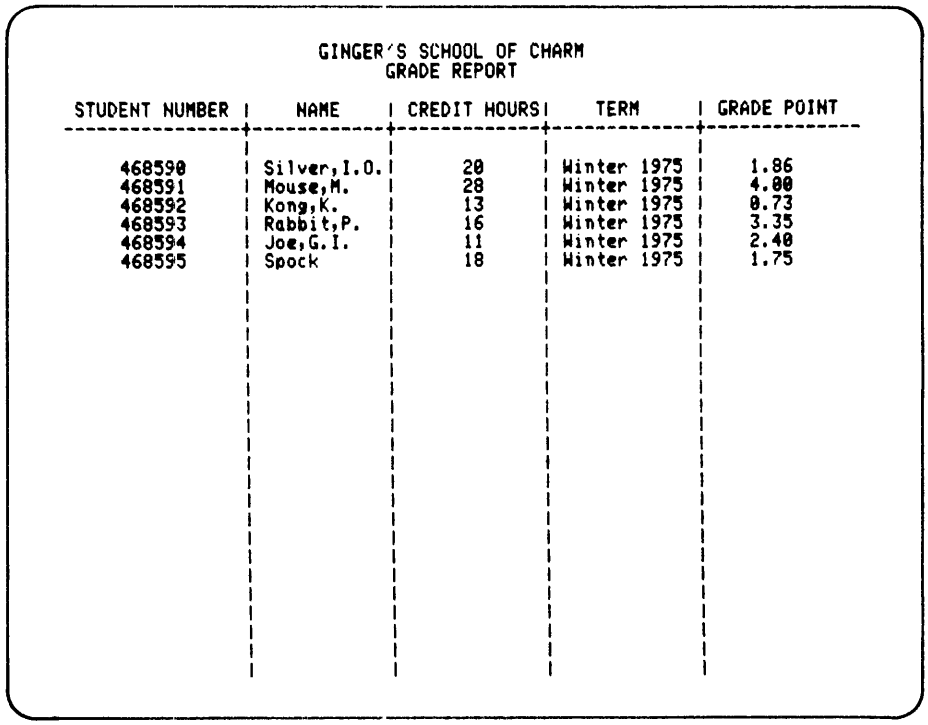
Inputting Logical Records from an ASCII Data File

After data items are arranged into logical records, converted to ASCII code, and stored on magnetic tape, a distinction between string data and numeric data cannot be made. For this reason, the INPUT statement is oriented toward inputting a complete logical record at a time. For example:

```
100  INIT
110  PAGE
120  PRINT "Which ASCII Data File Do You Wish to Access?"
130  INPUT F
140  FIND F
150  INPUT @33:A$,B$,C$,D$,E$,F$
160  GOSUB 1000
170  PRINT A$
180  PRINT B$
190  PRINT C$
200  PRINT D$
210  PRINT E$
220  PRINT F$
```

```
230  HOME
240  END
1000 REM —This routine prints the report card title and format—
1010 PRINT USING, "P,24T,24A":"GINGER'S SCHOOL OF CHARM"
1020 IMAGE 30T,12A,/,/,2T,14A,22T,4A,S
1030 PRINT USING 1020:"GRADE REPORT","STUDENT NUMBER","NAME"
1040 IMAGE 7T,14A,24T,4A,35T,11A,/,72"-","↑",2/
1050 PRINT USING 1040:"CREDIT HOURS","TERM","GRADE POINT"
1060 PRINT USING "31(17T"" | ""30T"" | ""44T"" | ""58T"" | ""/),""↑""5/":
1070 RETURN
```

GS Display Output



GINGER'S SCHOOL OF CHARM
GRADE REPORT

STUDENT NUMBER	NAME	CREDIT HOURS	TERM	GRADE POINT
468590	Silver,I.O.	20	Winter 1975	1.86
468591	Mouse,M.	28	Winter 1975	4.00
468592	Kong,K.	13	Winter 1975	0.73
468593	Rabbit,P.	16	Winter 1975	3.35
468594	Joe,G.I.	11	Winter 1975	2.40
468595	Spock	18	Winter 1975	1.75

Lines 100 and 110 in this program initialize the system and clear the GS display. Line 120 then asks the keyboard operator to enter the file number of the ASCII data file he wishes to access. Line 130 records the entry by assigning it to the variable F. Line 140 then positions the magnetic tape read/write head to the beginning of the specified file. This action opens the file for access.

In this example, the keyboard operator enters file number 2—the file containing the six student records shown in the illustration. For ease of illustration, this file was previously created using the following program:

```
100 INIT
110 PRINT "Which File for Storing the Data"
120 INPUT X
130 FIND X
140 DATA 468590,"Silver,I.O.",20,"Winter 1975",1.86
150 DATA 468591,"Mouse,M.",28,"Winter 1975",4
160 DATA 468592,"Kong,K.",13,"Winter 1975",0.73
170 DATA 468593,"Rabbit,P.",16,"Winter 1975",3.35
180 DATA 468594,"Joe,G.I.",11,"Winter 1975",2.4
190 DATA 468595,"Spock",18,"Winter 1975",1.75
200 IMAGE 6T,6D,19T,18A,37T,2D,46T,11A,63T,D.2D
210 FOR I=1 TO 6
220 READ A,B$,C,D$,E
230 PRINT @33: USING 200:A,B$,C,D$,E
240 NEXT I
250 END
```

In line 150, previous program, the INPUT statement is used to bring six student records back into memory. Because the INPUT statement is record orientated, each student record is assigned to a string variable. In this case, the entire record for I.O. Silver is assigned to A$. The entire record for M. Mouse is assigned to B$, and so on. It is important to remember here that the BASIC interpreter assigns all of the characters up to the first CR to A$, all of the characters up to the second CR to B$, and so on. Because CR is used as the delimiter between logical records, each logical record brought into memory is assigned to one string variable.

After the student records are brought into memory, line 160 is executed which transfers program control to line 1000, the beginning of a subroutine which prints a report card format on the GS display. (Refer to the PRINT statement and the IMAGE statement for an explanation of the techniques used in this subroutine.) After the subroutine is finished executing, program control is transferred back to line 170 where the student records are printed on the GS display (lines 170 through 220). Line 230 returns the alphanumeric cursor to the HOME position and line 240 terminates program execution.

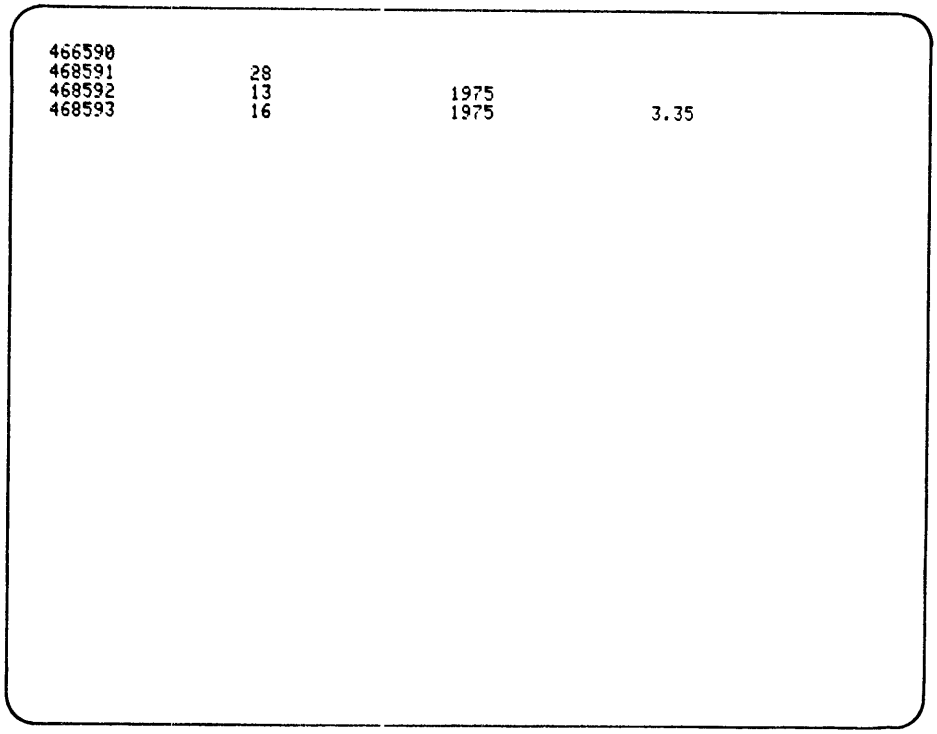
More can be done with the student records than just input them into memory and print them on the GS display, but program is terminated here because its only purpose is to show how to input logical records.

Inputting Numeric Fields from Logical Records

If an ASCII data file is accessed and numeric variables are specified in the INPUT statement instead of string variables, then only the numeric data in the logical record is brought into memory. The string data acts as a delimiter to each numeric value. For example:

```
100 INIT
110 FIND 2
120 INPUT @33:A
130 PRINT "L",A
140 INPUT @33:B,C
150 PRINT B,C
160 INPUT @33:D,E,F
170 PRINT D,E,F
180 INPUT @33:G,H,I,J
190 PRINT G,H,I,J
200 END
```

GS Display Output



This program inputs numeric data from the student records in file 2. Line 1ØØ initializes the system and line 11Ø positions the tapehead to the beginning of file 2. Line 12Ø inputs the first numeric value in the first logical record and assigns the value to the variable A. In this case, the student number for I.O. Silver is input, and assigned to A, then printed on the GS display in line 13Ø.

Since the INPUT statement is orientated toward operating on one logical record at a time, the BASIC interpreter assumes that the input operation on this first record is complete. When line 14Ø is executed, the tape head is automatically positioned to the beginning of the next logical record. The remaining information in the first logical record is skipped; in this case, Student Name, Credit Hours, Term, and Grade Point are skipped.

The first two numeric data items in the second record are assigned to the variables B and C in line 14Ø. In this case, the Student Number and Credit Hours for M. Mouse are assigned to B and C and printed on the GS display in line 15Ø. In lines 16Ø and 17Ø, the Student Number, Credit Hours, and Year for the Term for K. Kong are assigned to the variables D, E, and F, respectively, and printed on the GS display. Notice that originally, the Term (Winter 1975) was transferred to the magnetic tape as a character string; however, because the difference between character string digits and numeric digits can not be made after the conversion to ASCII code, the digits 1975 are treated as numeric digits when they are brought back into memory.

When lines 18Ø and 19Ø are executed, all four numeric items in the student record for P. Rabbit are brought into memory and printed on the GS display. The program then terminates in line 2ØØ.

It can be seen from this program that numeric data items contained in logical records can be pulled from each record and input into memory by specifying numeric variables in the INPUT statement. It can also be seen that each new INPUT statement starts at the beginning of a new logical record. (This applies to all INPUT operations including those involving external peripheral devices on the General Purpose Interface Bus.)

Here's a program which takes advantage of these characteristics. The program is designed to input the numeric data items for each student record in the file. The program then computes the average credit hours for the class, then the average grade point.

```
100 INIT
110 PAGE
120 FIND 2
130 INPUT @33:A,B,C,D
140 INPUT @33:E,F,G,H
150 INPUT @33:I,J,K,L
160 INPUT @33:M,N,O,P
170 INPUT @33:Q,R,S,T
180 INPUT @33:U,V,W,X
190 PRINT "AVERAGE CREDIT HOURS = ";(B+F+J+N+R+V)/6
200 PRINT "AVERAGE GRADE POINT = ";(D+H+L+P+T+X)/6
210 END
```

Line 100 initializes the system and line 110 clears the GS display. The beginning of the ASCII data file is found in line 120, then the numeric data items in each student record are input into memory and assigned to numeric variables (lines 130 through 180).

Once the numeric data is in memory, numerous calculations can be performed. In this program, the average credit hours for the class are computed and printed on the GS display in line 190. Then, the average grade point for the class is computed and printed on the GS display in line 200. The program is terminated in line 210.

This is a very simple example to show how specific fields of numeric data can be input from logical records and used for calculations. More elaborate calculations can be performed to compute, for example, student percentile rank, and the data can be used to draw histograms and distribution curves for the class.

Inputting Records from the Middle of an ASCII Data File

The following programs are included here to illustrate how to input the information from a logical record located in the middle of an ASCII Data File. These methods are by no means the only methods that can be used or the "best" methods that can be used.

Quite often you'll want to input one or more logical records which are located in the middle of an ASCII data file. Because ASCII data files are sequentially read, you'll need to position the magnetic tape read/write head to the beginning of the record you want before inputting the record. Here is a method that can be used for accessing the middle of an ASCII data file. The data file from the last program is used for convenience.

```
100  INIT
110  PAGE
120  PRINT "Which Student Record File Do You Want?"
130  INPUT F
140  FIND F
150  PRINT "Which Student Record Do You Wish to Start With:"
160  INPUT X
170  IF X=1 THEN 210
180  FOR I=1 TO X-1
190  INPUT @33:A$
200  NEXT I
210  PRINT "How many Student Records Do You Want to Input?"
220  INPUT R
225  GOSUB 1000
230  FOR S=1 TO R
240  INPUT @33:A$
250  PRINT A$
260  NEXT S
270  HOME
280  END
1000 REM —This routine prints the report card title and format—
1010 PRINT USING "P,24T,24A":"GINGER'S SCHOOL OF CHARM"
1020 IMAGE 30T,12A,/,/,2T,14A,22T,4A,S
1030 PRINT USING 1020:"GRADE REPORT","STUDENT NUMBER","NAME"
1040 IMAGE 7T,14A,24T,4A,35T,11A,/,72"-","↑",2/
1050 PRINT USING 1040: "CREDIT HOURS","TERM","GRADE POINT"
1060 PRINT USING "31(17T"" | ""30T"" | ""44T"" | ""58T"" | ""/),""↑""5/":
1070 RETURN
```

This program initializes the system in line 100, clears the GS display in line 110, then asks the keyboard operator to enter the file number of the file he wishes to access in line 120. The keyboard operator enters a file number and presses the RETURN key. The BASIC interpreter assigns the entry to the variable F in line 130, and positions the magnetic tape read/write head to the beginning of the file in line 140.

The program now asks the question "Which Student Record Do You Wish to Start With?" Here, the keyboard operator specifies which record he wants. For example, if he wants to start with the fifth logical record, then he enters 5 and presses the RETURN key; if he wants to start with the sixth logical record, he enters 6, and so on. The BASIC interpreter assigns the entry to the variable X in line 160 and makes a comparison in line 170. If the entry is 1, then the tape head is already in position, so program control advances to line 210.

If the entry is not 1, then the BASIC interpreter executes lines 18Ø through 2ØØ to position the read/write head to the beginning of the specified logical record.

This positioning technique uses a FOR/NEXT loop to advance the read/write head through the file. In line 18Ø, the ending value for the index (I) is specified as the numeric expression X−1. This means that if the keyboard operator enters 5 in response to the question in line 15Ø, the ending value for I is computed to be 4. The FOR/NEXT loop is repeated 4 times. Each time the loop is executed, a logical record is input and assigned to the variable A$ (line 19Ø). In this case, the first four records are input before program execution continues to line 21Ø. This action positions the read/write head to the beginning of logical record number 5. The fact that the first four logical records are input and overwritten in A$ is of no importance. Lines 17Ø through 2ØØ are in the program only to position the tape head to the beginning of the specified logical record.

After the tape head is properly positioned, the program asks the question "How many Student Records Do You Want to Input?" The BASIC interpreter records the entry in line 22Ø by assigning the value to the variable R. Program control then branches to the subroutine starting in line 1ØØØ and the Report Card format is printed on the GS display. Control is then returned to line 23Ø where the specified number of student records are input from the ASCII data file and printed on the GS display (lines 23Ø through 26Ø). In this example, if the keyboard operator enters 2 in response to the question in line 21Ø, then logical record number 5 and number 6 are input and printed on the GS display. After the records are printed, the cursor returns to the HOME position and program execution ends in line 28Ø.

This program illustrates a simple method for accessing an ASCII data file to bring logical records into memory. The problem with this method is, however, that you can't always remember the contents of the logical records in the file. Here's another program which allows you to locate student records by entering the student record number. The program uses string functions to examine each record for an identifying feature; in this case, the student number. When a match is made, the record is input into memory. Here's the program:

```
10Ø  INIT
11Ø  PAGE
12Ø  PRINT "Which Student File Do You Want?"
13Ø  INPUT F
14Ø  FIND F
15Ø  PRINT "Enter a Student Number and press RETURN"
16Ø  INPUT X$
17Ø  ON EOF (Ø) THEN 1Ø9Ø
18Ø  INPUT @33:A$
19Ø  P=POS(A$,X$,1)
2ØØ  Y$=SEG(A$,P,6)
```

```
210  IF X$< >Y$ THEN 180
220  GOSUB 1000
230  PRINT A$
240  HOME
250  END
1000 REM —This routine prints the report card title and format—
1010 PRINT USING "P,24T,24A":"GINGER'S SCHOOL OF CHARM"
1020 IMAGE 30T,12A,/,/,2T,14A,22T,4A,S
1030 PRINT USING 1020:"GRADE REPORT","STUDENT NUMBER","NAME"
1040 IMAGE 7T,14A,24T,4A,35T,11A,/,72"-","↑",2/
1050 PRINT USING 1040:"CREDIT HOURS","TERM","GRADE POINT"
1060 PRINT USING "31(17T"" | ""30T"" | ""44T"" | ""58T"" | """/),""↑""5/":
1070 RETURN
1080 STOP
1090 REM —This subroutine returns the program to line 140 on EOF—
1100 PRINT "Student Record Not Found—Try Again!"
1110 DELETE X$
1120 GO TO 140
1130 END
```

This program begins execution in a manner which is similar to the last program. Line 100 initializes the system environmental parameters, line 110 clears the GS display, and line 120 prints the message "Which Student File Do You Want?" Assuming the keyboard operator is working with the same ASCII data file, he enters 2 and presses the RETURN key. Line 130 records the entry by assigning the numeric constant 2 to the variable F. Line 140 then positions the magnetic tape read/write head to the beginning of file 2.

Lines 150 through 210 form a routine which searches for the logical record with the specified student number. The routine starts in line 150 by asking the keyboard operator to enter a student number and press the RETURN key. Line 160 inputs the number from the keyboard and assigns the number to the variable X$. Notice here that the number is treated as a character string instead of a numeric value because a string variable is specified in the INPUT statement.

Line 170 activates the End of File interrupt facility in preparation for the search. This is necessary because it specifies the course of action to be taken if the search fails to find the record. (The course of action will be explained in a moment.)

Line 18Ø begins the search by inputting the first logical record into memory and assigning the record to A$. Line 19Ø then examines A$ for the specified student number. In this case, the POS (Position) function is used and in effect gives the BASIC interpreter the following instructions.

1. Search the student record assigned to A$ for the student number assigned to X$, starting with the leftmost character (position number 1).

2. Return the starting position of the substring X$ and assign the value to the variable P.

3. If you can't find the student number, assign Ø to the variable P. (For details on the POS function, refer to the Character String section.)

In line 2ØØ, a copy of the six digit student number in A$ is made and assigned to Y$. The two numbers assigned to X$ and Y$ are compared in line 21Ø to see if there's a match. If the numbers are equal (a match), then program execution continues to line 22Ø. If they aren't equal, program control is transferred back to line 18Ø where the next logical record is input and compared with the student number entry. This process continues until a match is made or until the end of the file is reached.

Assuming a match is made in line 21Ø, program execution continues to line 22Ø where control is transferred to the subroutine starting in line 1ØØØ. This subroutine prints a report card format on the GS display, then returns control to line 23Ø, where the student record is printed on the GS display. The cursor returns to the HOME position in line 24Ø and program execution terminates in line 25Ø.

If the end of the file is reached without finding the student number and an attempt is made to input the End Of File character, then program control is transferred to line 17Ø, then to line 1Ø9Ø, where the message "Student Record Not Found—Try Again!" is printed on the GS display. The invalid student number assigned to X$ is deleted in line 111Ø and program control is transferred to line 14Ø where the magnetic tape head is positioned to the beginning of the file and a request is made for another student number entry.

Inputting Program Lines

If the internal magnetic tape is positioned to the beginning of an ASCII program file, and the program is not secret, then an INPUT statement specifying one or more string variables can be used to input program lines into memory. For example:

```
FIND 3
INPUT @33:A$,B$
```

When the statement FIND 3 is entered from the GS keyboard and the RETURN key is pressed, the internal magnetic tape head is positioned to the beginning of file 3. If file 3 is a non-secret program file and the statement INPUT @33:A$,B$ is executed next, the first program line in that file is brought into memory and assigned to A$. The second program line is assigned to B$. The contents of A$ can be examined by entering A$ from the GS keyboard and pressing the RETURN key; the same can be done to display B$.

This feature allows you to examine the contents of a BASIC program stored in an external file without first deleting the current BASIC program from memory. For example, the first few program lines can be REMARK statements explaining the purpose and important features of the program. These program lines can be brought into memory, assigned to string variables, and examined without disturbing the BASIC program currently in memory.

External Peripheral Devices

A peripheral device on the General Purpose Interface Bus can be selected as the input source for an INPUT operation by specifying the appropriate I/O address in the INPUT statement. For example:

```
250 INPUT @18: A,B,C$,D
```

When this statement is executed, the BASIC interpreter issues the I/O address @18,13: over the General Purpose Interface Bus. Primary address 18 tells peripheral device number 18 that it has been selected to take part in the upcoming I/O operation. Secondary address 13 is issued by default and tells device 18 to send the ASCII character string presently under its read/write head.

After the I/O address is issued, the BASIC interpreter gives control of the bus to the peripheral device and prepares to receive the ASCII character string (a logical record). The peripheral device sends the ASCII string over the GPIB and indicates the end of the transmission by sending a Carriage Return character or by activating the EOI signal line or both. The BASIC interpreter terminates the I/O operation by issuing the universal commands UNTALK and UNLISTEN over the GPIB.

Once the ASCII data string (a logical record) is received, the BASIC interpreter examines the string for the correct sequence of numeric data and string data according to the specified variable list. In this case, the ASCII data string should contain two numeric values followed by a character string. If the ASCII data string does not contain enough data or the correct type of data to assign values to the entire list of variables, then the BASIC interpreter re-addresses the peripheral device and asks for more data. The re-addressing procedure continues until all of the variables in the list have assigned values.

In this example, any non-numeric characters which precede the first numeric value are ignored. The first numeric value is assigned to the variable A. The second numeric value is assigned to the variable B. Any non-numeric characters in-between the two numeric items are ignored. The rest of the characters in the ASCII data string are assigned to C$. In this case, the variable D is left without an assigned value, so the BASIC interpreter re-addresses device 18 and asks for another logical record. The first valid numeric item in the second string is assigned to the variable D; the rest of the characters in the logical record are ignored. The BASIC interpreter then continues to the next statement in the BASIC program.

It is important to remember that the BASIC interpreter treats ASCII input from an external peripheral device the same as it treats ASCII input from the internal magnetic tape unit or the GS keyboard. If a string variable is specified, all of the characters up to the first Carriage Return are assigned to the string variable (unless an alternate delimiter is specified and appears in the ASCII data string before the CR). If a numeric variable is specified, the first valid number found after starting the search is assigned to the numeric variable. Numeric data must be sent most significant digit first in any valid format (standard notation or scientific notation.) For example, numeric data can take the form 111.222E-3, or .ØØØ4, or 124.34, or 1ØØØØØ. The first non-numeric character after the number acts as the delimiter.

Specifying Alternate Delimiters for INPUT Operations

If a percent sign (%) is specified in place of the "at" sign (@) in the I/O address for the INPUT statement, the BASIC interpreter uses a previously specified ASCII character for a record separator character and a previously specified ASCII character for an End Of File mark. This feature gives the Graphic System the ability to adapt its INPUT format requirements to the output formats used by different peripheral devices. The ASCII characters to be used as the alternate record separator and End of File mark are specified in a special PRINT statement.

The alternate delimiters and character to be deleted are selected by addressing the second processor status byte as follows:

```
PRINT @37,Ø:Ø—255,Ø—255,Ø—255
```

When this statement is executed (either directly from the GS keyboard or under program control) the alternate delimiters are established. Primary address 37 tells the microprocessor to prepare to receive information which represents a change in a processor status byte. Secondary address Ø tells the microprocessor that the parameters to the second status byte are to be changed. Three numbers separated by commas are then specified after the colon (:) in the PRINT statement. These numbers each represent the decimal equivalent of an ASCII character.

The first number after the colon represents the decimal equivalent of the record separator character to be used in the INPUT operation and must be in the range Ø–255. For example, if 65 is specified, the ASCII letter "A" is used as the record separator instead of CR.

The second number specified after the colon in the PRINT statement represents the End of File (EOF) character to be used and must be in the range Ø–255. For example, if 66 is specified as the second number, the first "B" found in the incoming ASCII data string is treated as an EOF mark. When a "B" is found, program execution is terminated and an EOF error message is printed on the GS display.

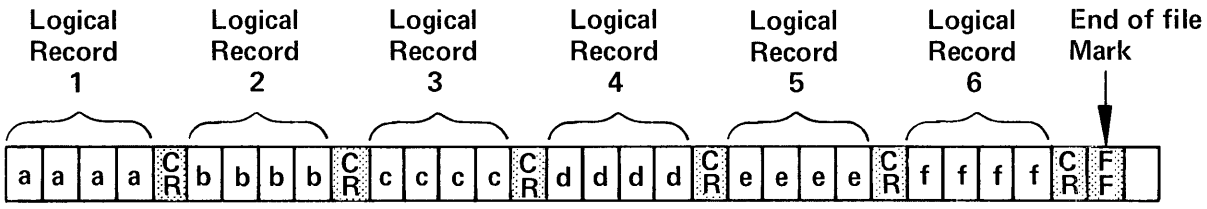
The third number specified after the colon indicates which ASCII character is to be deleted from the incoming ASCII data string. For example, if 67 is specified, the ASCII character "C" is deleted from the ASCII data string each time it appears. Again this number represents the decimal equivalent of an ASCII character. If the number specified as the third parameter is greater than 127, then every character is retained in the incoming data string.

Once these status byte parameters are set, the only way they can be changed is to execute another PRINT @37,Ø: statement or turn off the system power.

Specifying An Alternate Record Separator. Care must be taken when specifying an alternate record separator delimiter to ensure that parts of logical records are not lost. This can happen if the ASCII data string contains both Carriage Return characters and the alternate record separator character. The following examples illustrate how this happens:

Example 1—Normal Delimiting Action for INPUT operations.

```
5ØØ INPUT @2Ø:A$,B$,C$,E$,F$,G$
```



This example illustrates how normal delimiting action occurs during an INPUT operation. If the "at" sign (@) is specified, Carriage Return is the only valid record separator character and hexidecimal FF is the only valid End Of File character. The character strings shown above represent ASCII character strings received from peripheral device number 2Ø when statement

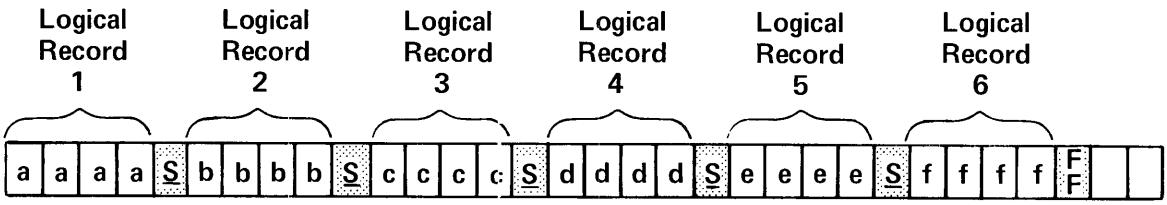
500 is executed. Each CR character marks the end of a logical record. The BASIC interpreter re-addresses peripheral device number 20 after each CR is received to tell it to send the next logical record. The logical record assignments are made as follows:

```
A$="aaaa"
B$="bbbb"
C$="cccc"
D$="dddd"
E$="eeee"
F$="ffff"
G$=End Of File Mark
```

When an attempt is made to assign the End of File mark (hexidecimal FF) to G$, a fatal error occurs, and program execution is aborted. The appropriate error message is printed on the GS display.

Example 2—Delimiting Action when the Alternate Record Separator is specified.

```
510 PRINT @37,0:19,255,255
520 INPUT %20:A$,B$,C$,D$,E$,F$,G$,
```



When line 510 is executed, the alternate record separator character is specified as ASCII decimal equivalent 19 (the DC3 control character). The End Of File character is specified as hexidecimal FF (255) and the character to be deleted from the ASCII data string is specified as 255 (no character to be deleted). Notice that when the microprocessor status parameters are set, all three parameters must be specified, regardless of whether they are changed from their last values or not.

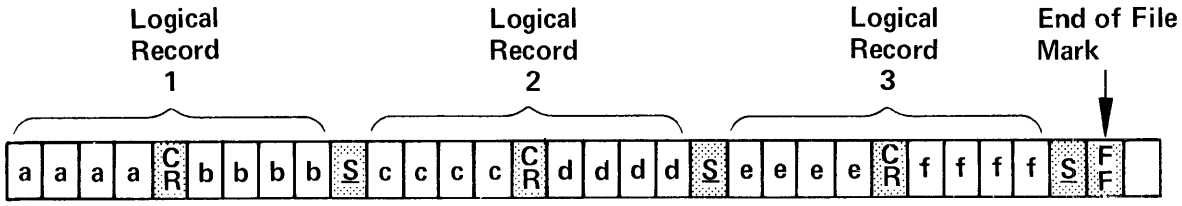
Line 520 inputs logical records from peripheral device 20 on the General Purpose Interface Bus. Because the percent sign (%) is specified in the I/O address instead of the "at" sign (@), the BASIC interpreter considers the DC3 control character (represented by the S symbol) as a valid record separator. The ASCII data string received from peripheral device 20 is shown above. The following assignments are made:

```
A$="aaaa"
B$="bbbb"
C$="cccc"
D$="dddd"
E$="eeee"
F$="ffff"
G$=End Of File Mark
```

Notice that the logical record assignments are the same as the previous example. Each time a DC3 control character is found, the BASIC interpreter treats the character as the end of a logical record. The BASIC interpreter re-addresses peripheral device number 2Ø after each DC3 is found and tells it to send the next logical record. This happens six times until an attempt is made to input the End Of File mark and assign it to G$. In this case, peripheral device number 2Ø uses the same End Of File character as the normal value (hexidecimal FF). When this character is received, a fatal error occurs and program execution is aborted. The appropriate error message is printed on the GS display.

Example 3—Intermixing Carriage Returns and the Alternate Record Separator Character.

```
53Ø PRINT @37,Ø:19,255,255
54Ø INPUT %2Ø:A$,B$,C$,E$,F$,G$
```



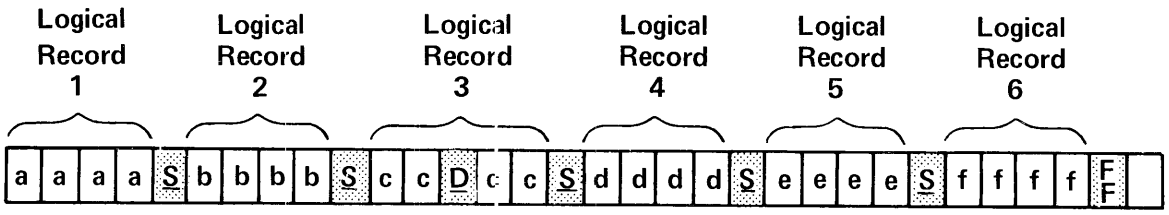
This example shows what happens when the Carriage Return character and the alternate Record Separator character are alternately used between logical records. The same program lines are re-executed that were used in the last example. This time, however, peripheral device number 2Ø sends the ASCII data strings shown in the illustration. Because the percent sign is specified in the INPUT statement, the BASIC interpreter assumes that only three records are sent over the bus, each terminated with the DC3 control character as shown in the illustration. The following assignments are made:

```
A$="aaaa"
B$="cccc"
C$="eeee"
D$=End Of File Mark
```

Because the Graphic System does not have the capability to handle the CR character as part of a character string, the string assignment for each logical record is terminated at the CR character. In this example, the characters "bbbb", "dddd", and "ffff" are lost from the ASCII data string.

Specifying an Alternate End of File Character. The second parameter in the PRINT statement specifies the alternate End Of File character. This parameter is specified as a decimal number between Ø and 255 and represents the decimal equivalent of an ASCII character. For example:

```
55Ø PRINT @37,Ø:19,4,255
56Ø INPUT %2Ø:A$,B$,C$,D$,F$,G$
```



In line 55Ø, the alternate record separator character is specified as decimal 19 (the DC3 control character) and the alternate End Of File character is specified as decimal 4 (the EOT control character). In line 56Ø, peripheral device 2Ø sends the ASCII string shown in the illustration. The assignments are made as follows:

```
A$="aaaa"
B$="bbbb"
C$="cc"
```

The first two logical records are assigned to A$ and B$, respectively. Notice that the alternate record separator (S) is used. When an attempt is made to assign the third logical record to C$, the alternate End Of File character is found. The characters in the third logical record up to the D character are assigned to C$. An error occurs when an attempt is made to input the D character, program execution is aborted, and the appropriate error message is printed on the GS display.

Specifying a Character to be Removed From the ASCII Data String. The third parameter specified in the PRINT statement tells the BASIC interpreter which character to remove from the incoming ASCII data string. If the parameter is specified as an integer from 1 to 127, the BASIC interpreter assumes the integer is the decimal equivalent of an ASCII character and removes that character each time it appears in the ASCII data string. If this parameter is specified as an integer greater than 127, but less than 256, then all of the characters are retained in the incoming ASCII data string.

Specifying the Microprocessor Status Parameters As Numeric Expressions. Each of the three microprocessor status parameters can be specified as numeric expressions and set under program control as long as each numeric expression can be reduced to a numeric constant between Ø and 255. Any numeric constant outside this range results in an error, program execution is aborted, and a system error message is printed on the GS display.

THE KILL STATEMENT

Syntax Form:

[Line number] KIL [I/O address] numeric expression

Descriptive Form:

[Line number] KILL [I/O address] tape file number

Purpose

The KILL statement "wipes out" the specified magnetic tape file. Once killed, the tape file cannot be recovered.

Explanation

The Internal Magnetic Tape Unit

The KILL statement causes the internal magentic tape to execute a high speed search for the specified magnetic tape file. Once found, the file header is marked NEW and the information in the file cannot be recovered (unless the magnetic tape status is changed to non-header format). For example:

```
11Ø KILL 5
```

When this statement is executed under program control, the internal magnetic tape unit executes a high speed search for file 5. When file 5 is found, the file header is marked NEW. This makes the file available for reassignment. The old information in file 5 is not destroyed, but it cannot be accessed. The old information is automatically overwritten when new information is sent to the file. This information can be a BASIC program, ASCII data, or binary data.

The tape file number can be specified as a numeric expression as long as the expression can be reduced and rounded to a positive integer; the integer must be a valid magnetic tape file number.

If a KILL statement is executed under program control and the specified tape file doesn't exist, then the tape is rewound and program execution is aborted. The appropriate error message is printed on the GS display.

External Peripheral Devices

The KILL statement can be directed toward an external peripheral device by specifying the appropriate I/O address in the statement. For example:

```
120 KILL @12:8
```

When this statement is executed, the BASIC interpreter issues the I/O address @12,7: over the General Purpose Interface Bus (GPIB). Primary address 12 tells peripheral device number 12 that it has been selected to take part in the upcoming I/O operation. Secondary address 7 is issued by default and tells peripheral device 12 that the information it is about to receive is the file number of a file to be killed. The number 8 is then issued to device 12 over the GPIB as an ASCII character string terminated with a Carriage Return (CR). It is up to device number 12 to interpret the number 8 as the file to be killed and then kill the file.

THE LINK ROUTINE

Syntax Form:

[Line number] CALL { "LINK," / string variable, } [I/O address;] line number

Descriptive Form:

[Line number] CALL routine name, [I/O address;] line number of entry point

Purpose

The LINK routine transfers a stored binary program to memory from the specified peripheral device without disturbing variables and associated values stored in memory.

Explanation

The LINK routine erases the program currently in memory, but retains all variables and their currently assigned values. Then, a binary program is loaded into memory from the specified input device. The BASIC line counter is set to the starting line number and execution starts at that point.

If an input device is not selected, the internal magnetic tape unit is selected by default.

The LINK routine allows you to break a large BASIC program into subprograms, store them on tape, and then structure the flow of execution by using the LINK routine. By using the LINK routine, the size of a program is no longer limited by the storage capacity of memory. You can break a program into sections and load and execute the sections.

To use the LINK routine, the read/write head of the peripheral device must first be positioned at the beginning of a binary program file.

To locate the beginning of a binary program file, use the FIND statement. After the LINK routine is executed, variables and values are kept and the current program is erased. The entire new binary program (previously located by the FIND statement) is then loaded into memory.

If the LINK routine is executed in immediate mode, execution stops after the program is transferred. The loaded program can then be executed by entering a RUN statement and pressing the RETURN key. Execution begins with the starting line number.

If the LINK routine is executed under program control, then after the binary program is transferred, execution automatically begins with the starting line number.

If a LINK routine brings a SECRET BINARY program into memory, then the entire contents of memory are made secret. As always, secret programs can only be executed.

EXAMPLES

To use the LINK routine a BASIC program must be stored in binary format. As an example, the following program is stored in file 1 on the internal magnetic tape unit. Since this file is stored in binary format, the file is retrieved by using the BOLD routine. A LIST statement is executed to show the program.

```
FIND 1
CALL "BOLD"
LIST
100 INIT
110 A$ = "CHANGE"
120 PRINT "SOME THINGS NEVER";A$
```

Now, the program is executed.

```
RUN
SOME THINGS NEVER CHANGE
```

This binary program is used in the following examples by the LINK routine:

```
100 A$ = "STAY THE SAME"
110 FIND 1
120 CALL "LINK", 120
```

In line 100, A$ is given the value "STAY THE SAME". The binary file is then located by the FIND statement. The LINK routine directs execution to begin in line 120 of the binary program. So . . .

```
RUN
SOME THINGS NEVER STAY THE SAME
```

The stored binary program is now in memory. No statements have been changed. A$ retains the value "STAY THE SAME" because execution started in line 120.

Notice the INIT statement in line 100 of the binary program. If the LINK routine is specified by

```
CALL "LINK",100
```

then execution would begin with line 100. The INIT statement puts all variables in an undefined state, including any variables retained by the LINK routine. A$ would be undefined. Then in line 110 of the binary program, A$ would be set to "CHANGE".

If the LINK routine is specified by

```
CALL "LINK",110
```

then A$ is retained with the value "STAY THE SAME", but is set to "CHANGE" in line 110 of the binary program.

From these examples, you can see how execution and program flow can be controlled by using the LINK routine.

Specifying a Peripheral Device

The LINK routine can use any peripheral tape drive in the Graphic System storing a binary program. You can specify the input device in this way:

```
FIND @ 2:8
CALL "LINK",2;150
```

In this case, file number 8 on device number 2 is selected as the input device on the General Purpose Interface Bus. Execution begins with line 150.

THE MARK STATEMENT

Syntax Form:

[Line number] MAR [I/O address] numeric expression , numeric expression

Descriptive Form:

[Line number] MARK [I/O address] number of files , number of bytes per file

Purpose

The MARK statement creates the specified number of files on magnetic tape. The files are created on the internal magnetic tape if an external magnetic tape device is not specified.

Explanation

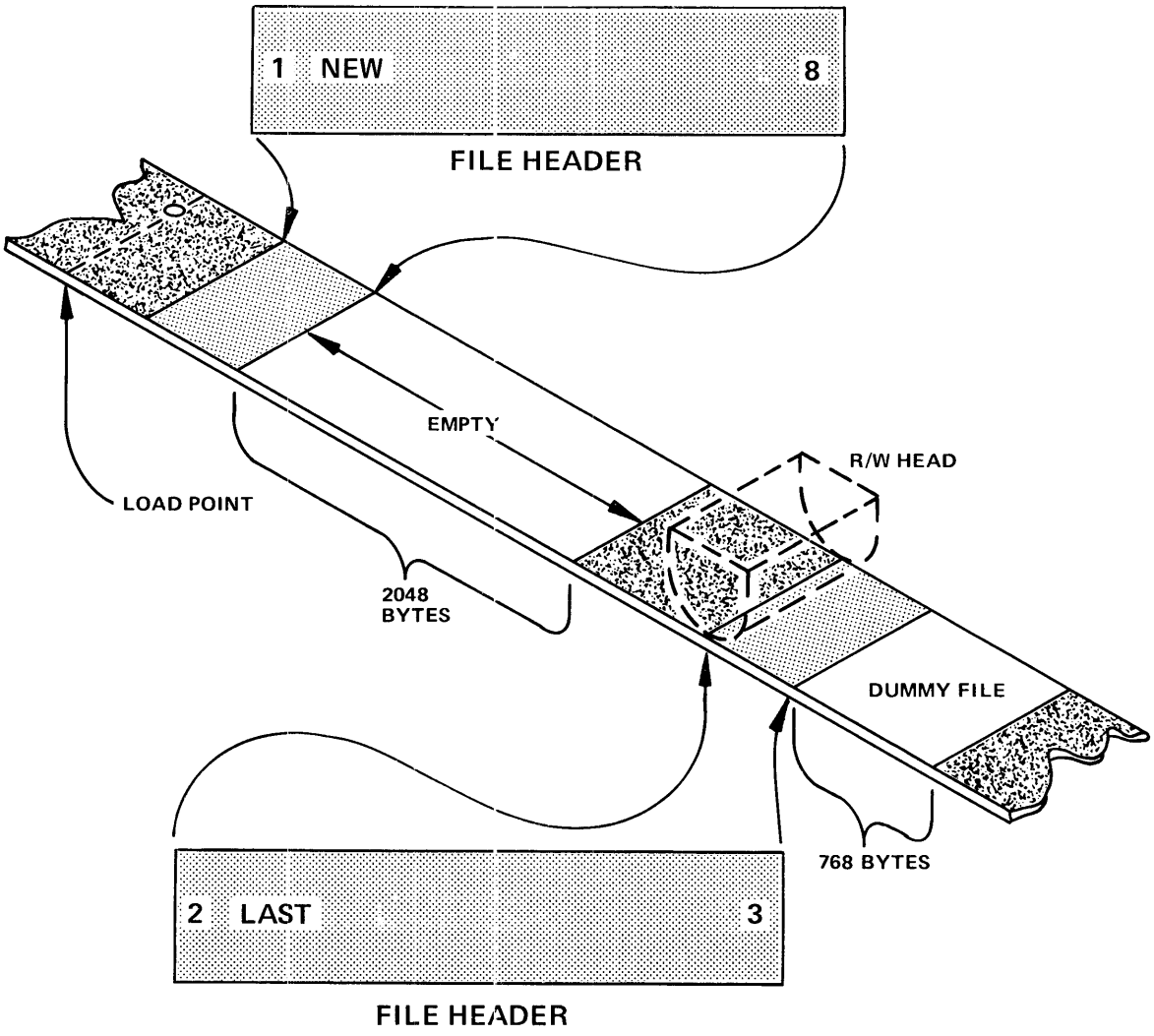
Magnetic Tape Files

Before programs and data can be stored on magnetic tape, the tape must be marked into files. Each file serves as a storage area to hold either a BASIC program or data.

New files are created on magnetic tape as follows:

```
100 FIND 0
110 MARK 1,2000
```

When line 100 is executed, the tape rewinds to the load point (the beginning). Line 110 creates one file which is approximately 2000 bytes long. This action is like a carpenter placing pencil marks on a board as he measures the board with a yardstick. The results are shown in the following diagram.



The first file is automatically labeled file number 1 and starts approximately one inch past the load point. The file is marked off in 256 byte increments called physical records. The first physical record serves as the file header for storing information about the file. This information includes the file number, the file length, the file contents (program or data), and the storage format (ASCII code or binary code). When a file is new (just created) the file header is marked NEW, as shown in the illustration.

The file storage area begins with the second physical record. The storage area is marked off in 256 byte increments. In this case, the storage area is specified as 2000 bytes in the MARK statement. Using the 256 byte yardstick, the BASIC interpreter creates a storage area 2048 bytes long. This area is made large enough to hold the specified number of bytes while

maintaining the 256 byte increment requirement. A three inch file gap is placed at the end of the storage area to mark the physical end of the file.

Approximately three inches past the end of the first file, a dummy file of minimum length (three physical records) is automatically created. This file is used to mark the beginning of the blank portion of the tape and, in this case, the file is labeled file number 2. When it comes time to create additional files on the tape, executing a FIND 2 statement positions the tape head to the beginning of the dummy file header as shown in the illustration. The tape head is also automatically positioned there after each MARK statement. Executing a MARK statement overwrites the dummy file with a new file which can be used to store information. (The dummy file can not be used to store information).

Notes about Marking Files

Up to 255 files can be created with a single MARK statement. Each file is marked to the specified length. For example, MARK 5,5ØØØ creates 5 files containing 5ØØØ bytes each. If the tape is positioned at the load point when this MARK statement is executed, the five files are numbered 1 through 5. The dummy file is marked file number 6. Another MARK statement executed immediately afterwards creates additional files. If the statement MARK 2,1ØØØ is executed with the tape head positioned on dummy file 6, 2 more files are created with 1ØØØ bytes each. The new files are labeled number 6 and number 7. A new dummy file is created and labeled number 8.

The second parameter in the MARK statement specifies the file length in bytes. The term "bytes" here is equivalent in usage to the term "bytes" when referring to memory space in the random access memory. If a BASIC program takes up 5ØØØ bytes of memory, for example, then a 5ØØØ byte file is usually large enough to store the program. The second parameter can be specified as a numeric expression as long as the expression reduces to a numeric constant within the range ± 1.67E+7, excluding Ø. If the number is negative, the absolute value of the number is used.

If an old magnetic tape is rewound to the beginning and a MARK statement is executed, the new files are marked on the tape. The old information underneath the new files and any information stored on the tape beyond the new files is lost.

If an old magnetic tape contains 1Ø files, for example, and the statements FIND 5 and MARK 1,2ØØØ are executed, then file 5 is remarked as a new file 2ØØØ bytes long. The information previously stored in files 5 through 1Ø is lost forever.

What to do after Marking a File

After one or more new files are created with the MARK statement, the tape head finishes at the beginning of the dummy file header as shown in the previous illustration. To store information in a file, a FIND statement must first be executed to position the tape head to the beginning of the storage area and open the file for access. For example, if five new files are created on a new tape, then the statement FIND 3 positions the tape head to the beginning of the storage area for file 3. (A TLIST statement can be executed to list the contents of each file on the tape. See TLIST in this section for details.)

A new file can be used to store a BASIC program or data, but not both in the same file.

A copy of the BASIC program currently in memory is sent to a file with the SAVE or CALL "BSAVE" statement. A program is brought back into memory with the OLD, CALL "BOLD", or CALL "LINK" statement.

The PRINT statement sends data items to the magnetic tape formatted in ASCII code. The data items transferred in a PRINT statement are treated as a single unit on magnetic tape, a logical record. Logical records can be any combination of numeric data and character strings. Logical records stored on magnetic tape in ASCII code are brought back into memory with the INPUT statement. (Refer to the PRINT and INPUT statements in this section for details.)

The WRITE statement sends data items to a magnetic tape file in binary format. This code is the same code used by the Graphic System to store data in the Random Access Memory. Data items stored in binary format are brought back into memory with the READ statement. (Refer to the WRITE statement and the READ statement in this section for details.)

Marking the Tape on an External Peripheral Device.

The MARK statement can be used to mark the tape on an external peripheral device by specifying the appropriate I/O address after the keyword MARK. For example:

```
120 MARK @10:2,2000
```

When this statement is executed, the I/O address @10,28: is sent over the General Purpose Interface Bus. Primary address 10 tells peripheral device number 10 that it has been selected to

take part in an I/O operation. Secondary address 28 is issued by default and tells device 1Ø to prepare to receive the parameters of a MARK statement. The parameters 2,2ØØØ are then converted into an ASCII character string and sent to device 1Ø over the GPIB. It is up to device 1Ø to receive the ASCII character string and mark the files according to the guidelines specified by the parameters in the string. The BASIC interpreter terminates the transfer by issuing the universal commands UNTALK and UNLISTEN over the GPIB.

Changing the Tape Format for Marking Operations

Internal magnetic tape parameters can be changed to give different file marking formats. For example, files can be created with or without a file header; they can be marked in 128 byte physical records or 256 byte physical records; and they can be marked with or without using the "Checksum" error checking technique. This facility allows the Graphic System to make digital tape recordings in a format which is compatible with other recording devices—a Tektronix 4923 Digital Cartridge Tape Recorder, for example. (Refer to the Magnetic Tape Status explanation in the Environmental Control section for details).

THE MTPACK ROUTINE

Syntax Form:

[Line number] CALL { "MTPACK" / string variable }

Descriptive Form:

[Line number] CALL routine name

NOTE

This routine is not available in the 4051 Graphic System.

Purpose

The MTPACK routine advances the magnetic tape in the internal tape drive to the end of the tape, then rewinds to the beginning of the tape.

Explanation

Because of the sequential nature of magnetic tape files, it is possible (by repeatedly accessing files at, for example, the beginning of the tape) to cause uneven tape tension between two locations on the tape. Also, mishandling may change the alignment of the tape with respect to the spool. This routine adjusts tension and alignment over the length of the tape.

The routine should be used before writing on a new tape or on a tape which has been dropped or subjected to severe temperature changes.

The routine takes about 1 minute to execute.

THE OLD STATEMENT

Syntax Form:

[Line number] OLD [I/O address]

Descriptive Form:

[Line number] OLD [I/O address]

Purpose

The OLD statement loads a BASIC program into the Random Access Memory from the specified input source. If an input source is not specified, then the internal magnetic tape unit is selected as the input source by default. An INIT and a DELETE ALL are performed before loading the program.

Explanation

Normally, an OLD program is a program which is previously saved under the SAVE statement, but doesn't have to be. Any program can be loaded into memory from a peripheral device as long as the program is compatible with the Graphic System BASIC language.

Loading a Program from the Internal Magnetic Tape Unit

If an OLD statement is executed without specifying an input source, the internal magnetic tape unit is selected as the input source by default. Before the OLD statement is executed, however, the readhead is normally positioned at the beginning of a program file. For example, the sequence...

```
FIND 7
OLD
```

loads the BASIC program stored in file number 7 into memory. The statement FIND 7 positions the tape head to the beginning of file 7. When the OLD statement is executed, the entire memory is cleared (current program and variables); the program from file number 7 is then brought into memory. As the program is brought in, the syntax of each statement is checked. If the syntax of a statement is incorrect, then the OLD operation is terminated and the incorrect

statement is printed on the GS display with the appropriate error message. The syntax of an incorrect statement can be corrected and entered into memory from the GS keyboard. The rest of the program can then be loaded using the APPEND statement, by entering a dummy target statement (5ØØ REM for example) from the keyboard, then executing an APPEND 5ØØ statement. (See APPEND in this section for details.)

If an OLD statement is executed under program control, a RUN statement is automatically executed after the program is brought into memory.

Specifying an External Peripheral Device as the Input Source

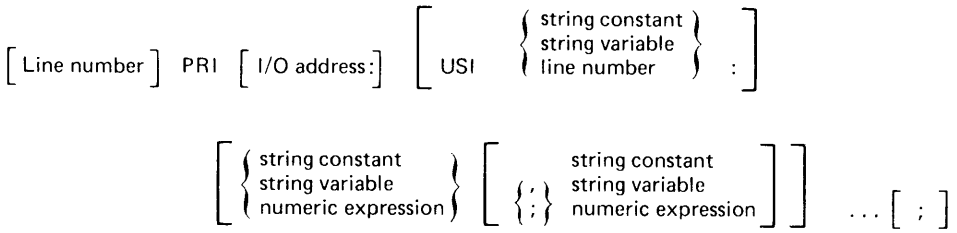
Any external peripheral device in the system can be specified as the input source for an OLD operation by specifying the appropriate I/O address. For example:

```
OLD @22:
```

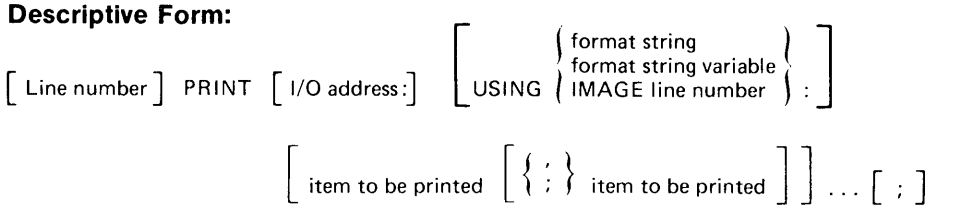
When this statement is executed, the BASIC interpreter loads a program from peripheral device number 22. Only the primary address of the peripheral is required because the BASIC interpreter automatically issues the secondary address 4 by default. This secondary address tells the peripheral device to send the program currently stored under the present position of its read head.

THE PRINT STATEMENT

Syntax Form:



Descriptive Form:



Purpose

The PRINT statement outputs data items to a specified peripheral device in the form of an ASCII character string. If a peripheral device is not specified, then the ASCII character string is sent to the GS display by default.

Explanation

The keyword PRINT would be appropriately called "OUTPUT ASCII DATA." In essence, the PRINT statement converts the specified data items into an ASCII character string and outputs the character string to the specified peripheral device. The conversion is made according to guidelines which are specified in the PRINT USING format string. If a PRINT USING format string is not specified, then the conversion is made according to a default print format.

Like all I/O statements, the PRINT statement provides for an optional I/O address. This address, if specified, selects a peripheral device to receive the ASCII character string. If an I/O address is not specified, then the I/O address @32,12: is issued by default. Primary address 32 refers to the GS display. Secondary address 12 tells the display to prepare to receive an ASCII character string and to print the character string on the display starting at the present position of the alphanumeric cursor.

The PRINT statement is the most powerful and the most complex statement in the Graphic System BASIC language. By using the unique I/O addressing facility of the language, virtually every BASIC statement involving ASCII data output can be duplicated with a PRINT statement. This includes magnetic tape statements like FIND, MARK, and SAVE, as well as display statements like MOVE and DRAW. The binary output statement WRITE cannot be duplicated.

Because of the complexity involved in explaining all the variations of this powerful statement, the following text is divided into topics. The topics begin with examples of the simple features of the PRINT statement and proceed to the more complex variations. Each example illustrates a specific feature of the PRINT statement.

The Default PRINT Format

If a PRINT USING format string is not specified in a PRINT statement, the following guidelines are used to convert the specified data items into a ASCII character string:

1. If a numeric expression is specified as a data item, the numeric expression is evaluated and reduced to a numeric constant.

2. If a numeric variable is specified as a data item, the numeric variable is replaced by its assigned value. If the variable doesn't have an assigned value, an error occurs and program execution is aborted.

3. If a string variable is specified as a data item, the string variable is replaced by its assigned value.

4. If a comma is used to separate two data items, spaces are inserted so that each data item fills an 18 character print field.

5. If a semicolon is used to separate two data items, additional spaces are not inserted between the data tems.

6. After the above operations are executed, the data items are converted into one ASCII character string and a Carriage Return character is added to the end of the of the ASCII string. The ASCII string is then shipped off to the specified peripheral device. The GS display automatically converts Carriage Return (CR) into Carriage Return/Line Feed (CR/LF).

7. If a semicolon is specified at the end of the PRINT statement, the Carriage Return character is not added to the end of the ASCII string.

The following examples illustrate the different features of the default print format.

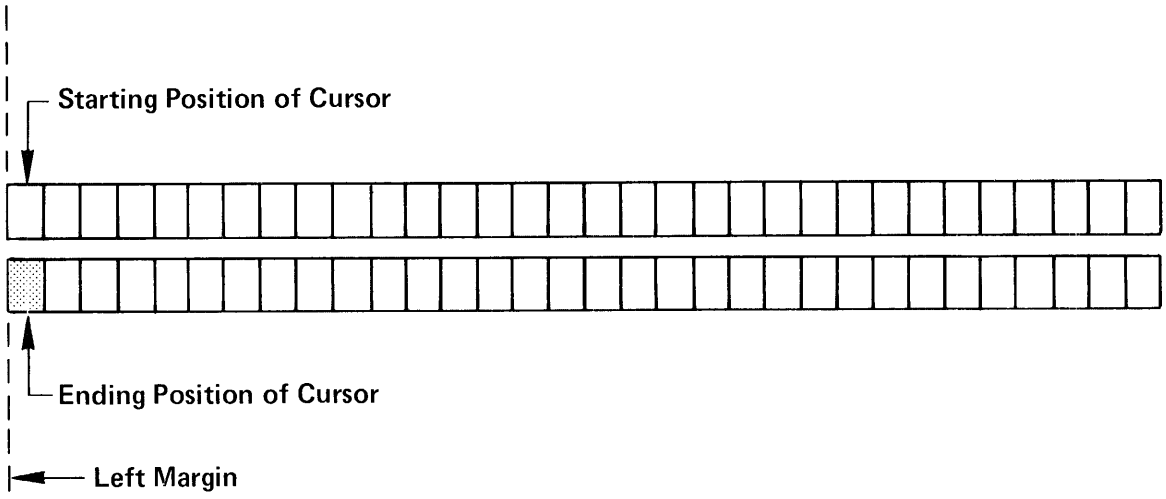
Printing a Blank Line

Executing a PRINT statement without parameters outputs a Carriage Return to the specified peripheral device. This drops the alphanumeric cursor down one line on the GS display and has the same effect as printing a blank.

Example 1—Printing a Blank

```
100 PRINT
```

GS Display Output



In this example, statement 100 is executed with the alphanumeric cursor positioned at the left-hand margin on the GS display (character position 1). Because data items are not specified in the PRINT statement, only a Carriage Return character (CR) is sent to the display. This moves the cursor down one line as shown in the illustration. If the cursor is on line 35 when the PRINT statement is executed (bottom line), a page full condition results when the Carriage Return is executed and a blinking "F" appears in the upper left-hand corner. The HOME/PAGE key must be pressed on the GS keyboard or the PAGE or HOME statement executed under program control before program execution continues. (This holds true only if the PAGE FULL parameter is set to the default value. See PAGE FULL in the Environmental Control section for details.)

Printing a Numeric Constant

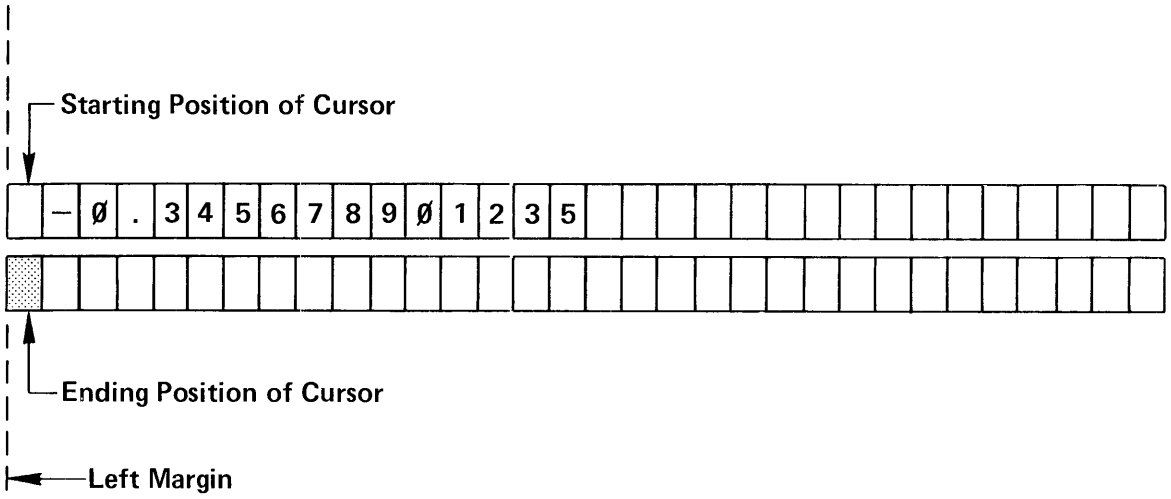
When a numeric constant is specified in a PRINT statement, the number is printed as specified unless it has any of the following properties:

- The number is between −1 and 1 and has more than 14 character positions (including the decimal point).
- The number is not between −1 and 1 and has more than 13 character positions (including the decimal point).
- The number is greater than ten million and in standard notation.

Example 2—Printing a Numeric Constant between −1 and 1.

```
120 PRINT -.34567890123456
```

GS Display Output

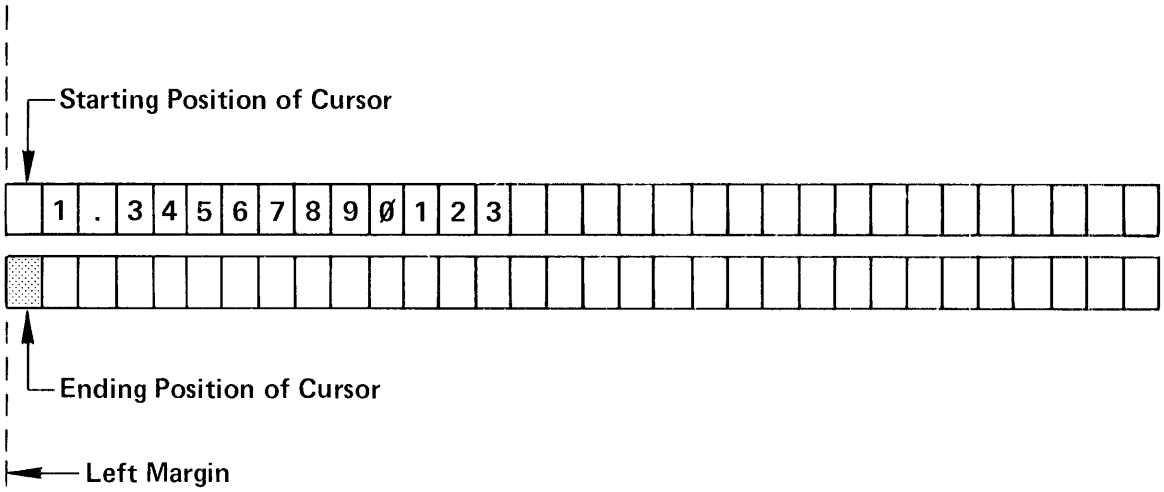


When the numeric constant in line 120 is printed, a zero is added to the left of the decimal point because a zero is not specified in the statement. Also notice that fourteen character positions are displayed (including the decimal point) because the number is between −1 and 1. This is done automatically to increase the display accuracy of small numbers.

Example 3—Printing a Numeric Constant Not Between −1 and 1

```
110 PRINT +1.345678901234
```

GS Display Output

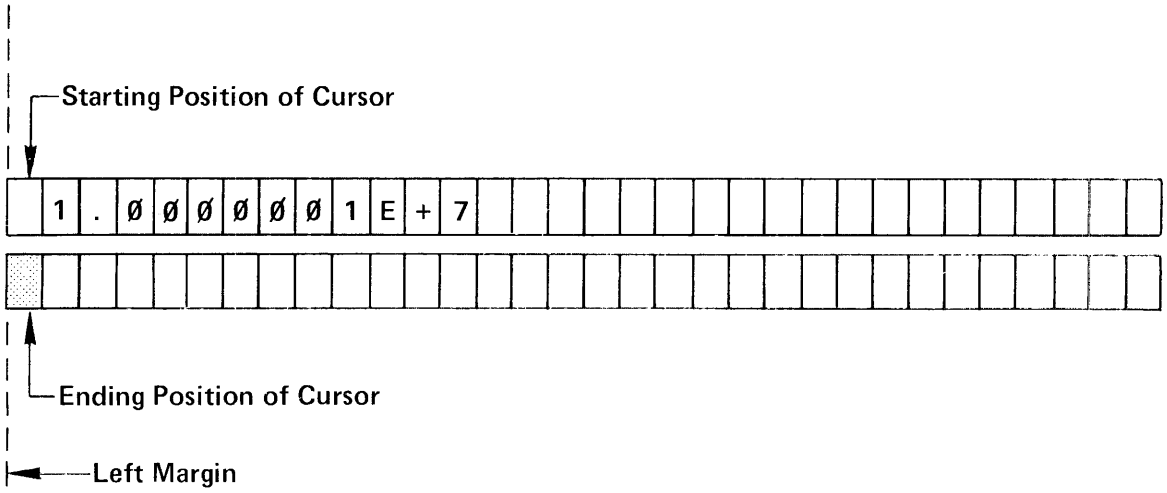


This example shows how the cursor moves over one space before a numeric constant is printed. Actually, when the number is converted into an ASCII character string, a space character is inserted as the first character in the string. This example further illustrates how a number greater than 1 is rounded to thirteen character positions if more than thirteen character positions are specified. Also notice that the plus sign (+) is suppressed and the automatic addition of a Carriage Return character returns the display cursor to the left-hand margin.

Example 4—Printing a Numeric Constant Greater than Ten Million

```
130 PRINT 10000001
```

GS Display Output

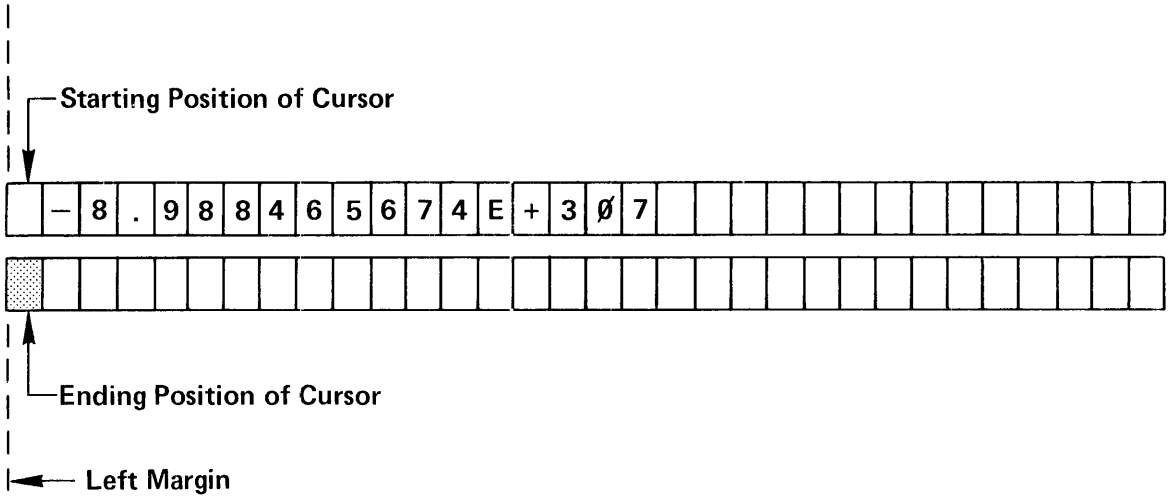


This example illustrates how a number greater than ten million is printed if specified in standard notation. The number is converted to scientific notation before it is sent to the GS display or to an external peripheral device.

Example 5—Printing a Number in Scientific Notation

```
14Ø PRINT -1.23456789Ø1234E 4ØØ
```

GS Display Output



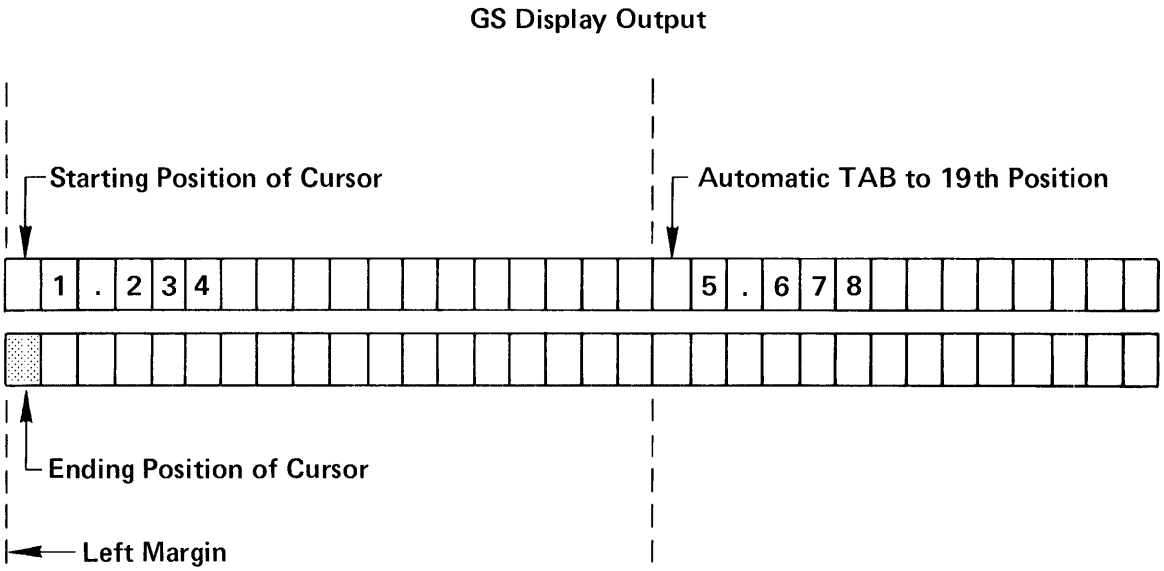
This example illustrates three points about printing a numeric value in scientific notation. First, if the specified number is outside the numeric range of the system, then the number closest to the range boundary is printed. Second, the decimal accuracy of a number written in scientific notation is not displayed to more than nine digits past the decimal point. Third, a plus sign (+) or minus sign (–) is automatically placed after the E.

An Automatic TAB is Specified by a Comma

If more than one numeric constant is specified in a PRINT statement, then the numeric constants must be separated by a comma or a semicolon. The comma causes each numeric constant to be printed in a separate 18 character field on the GS display. Unfilled character positions are padded with blanks.

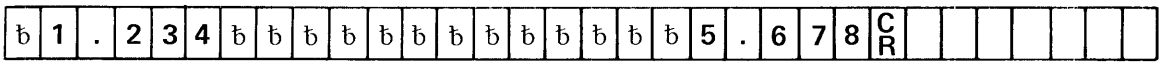
Example 6—Printing Numeric Constants Separated by Commas

```
150 PRINT 1.234,5.678
```



In this example, the number 5.678 is printed on the display starting at character position 19. The number appears to start in column 20, remember however that a space character is inserted before each number in the printout.

The automatic TAB feature is a function of the default print format only and is used in the ASCII conversion process. The automatic TAB is not a function of the GS display. When the parameters of the PRINT statement are converted into an ASCII character string, space characters are inserted so that each numeric value fills an 18 character field. In this example, the parameters 1.234,5.678 are converted into the following ASCII character string. The ƀ symbol represents the ASCII SPACE character.



If more than four data items are specified in a PRINT statement, and the data items are separated by commas, then the GS display prints one data item in each remaining print field in the current line, executes a Carriage Return at the right margin, and prints the remaining data items on the next line.

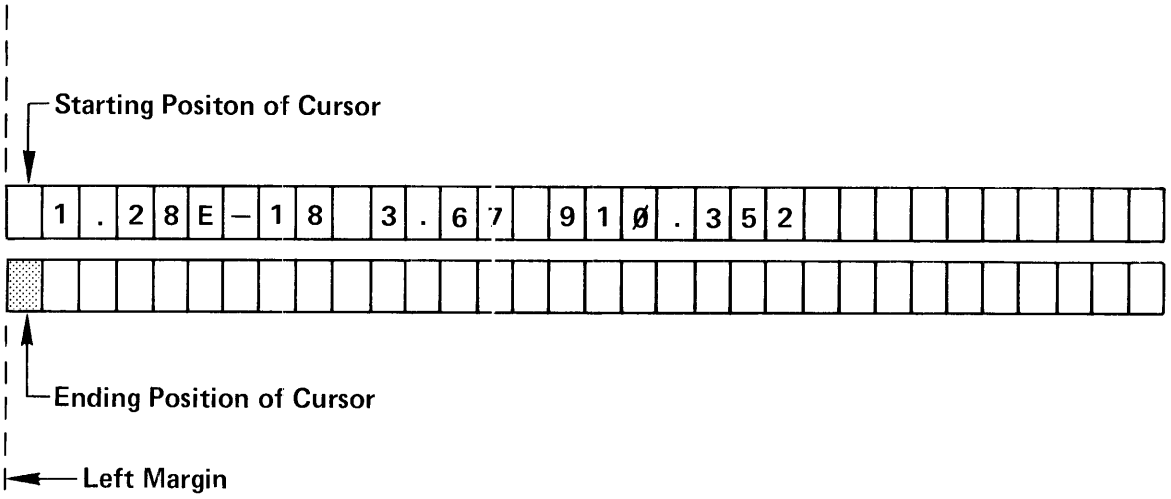
Suppressing the Automatic TAB Feature with a Semicolon

If data items are separated with semicolons in a PRINT statement, then the automatic TAB feature is suppressed.

Example 7—Suppressing the Automatic TAB Feature

```
160 PRINT 1.28E-18;3.67;910.352
```

GS Display Output



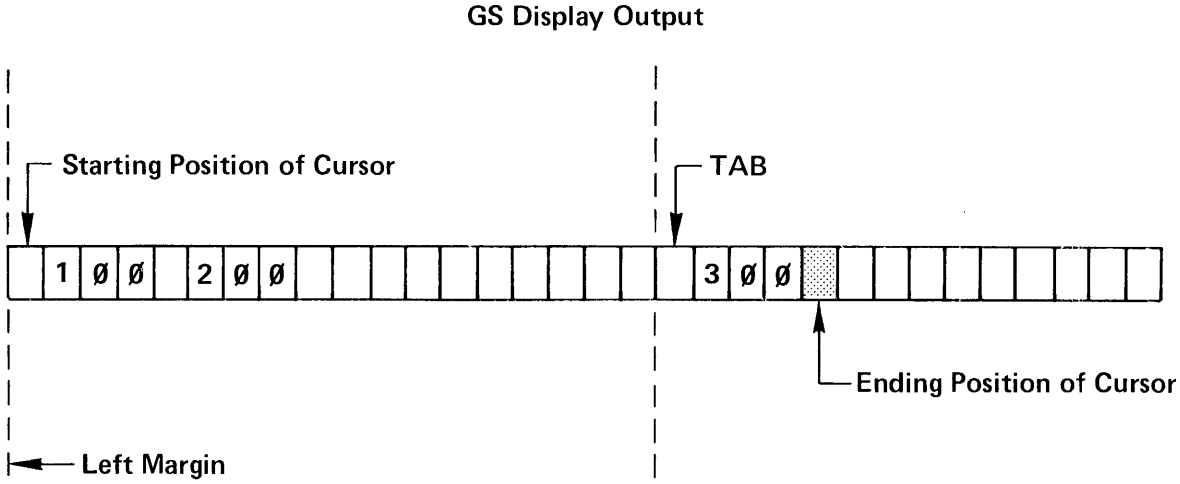
In this example, it is apparent why a space character is automatically inserted into the ASCII string before each numeric constant. If it wasn't automatically done in this example, then all the numbers would run together.

Suppressing the Automatic Carriage Return

If a semicolon (;) is specified at the end of a PRINT statement, the automatic addition of a Carriage Return is suppressed. The cursor remains in the position just beyond the last printed character.

Example 8—Suppressing the Automatic Addition of a Carriage Return

```
17Ø PRINT 1ØØ;2ØØ,3ØØ;
```



This example shows the results of specifying a semicolon at the end of a PRINT statement. The cursor finishes in the position just beyond the last printed character. This example also shows how commas and semicolons can be used in any combination to separate data items in the same PRINT statement.

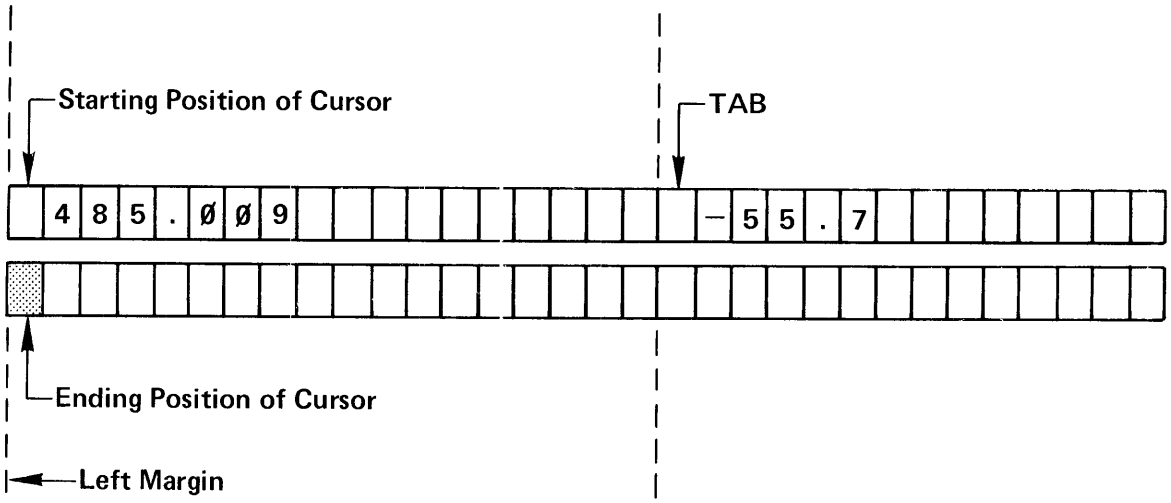
Printing a Numeric Variable

If a numeric variable is specified as a data item to be printed, then the numeric value currently assigned to that variable is printed. If the variable symbol does not have an assigned value, an error occurs and program execution is aborted.

Example 9—Printing a Numeric Variable

```
18Ø LET X = 485.ØØ9
19Ø PRINT X,-55.7
```

GS Display Output



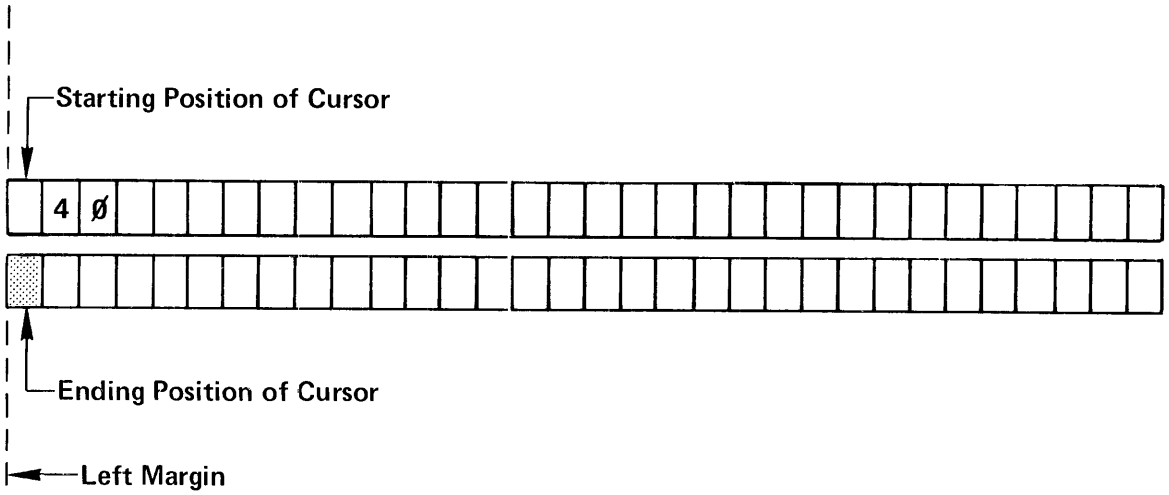
Printing a Numeric Expression

If a numeric expression is specified as a data item, the numeric expression is evaluated and reduced to a numeric constant before it is printed. If the specified numeric expression can't be reduced to a numeric constant (if it contains an undefined variable for example) then an error occurs and program execution is aborted.

Example 1Ø—Printing a Numeric Expression

```
2ØØ LET X = 5
21Ø PRINT X↑2+2* X+5
```

GS Display Output



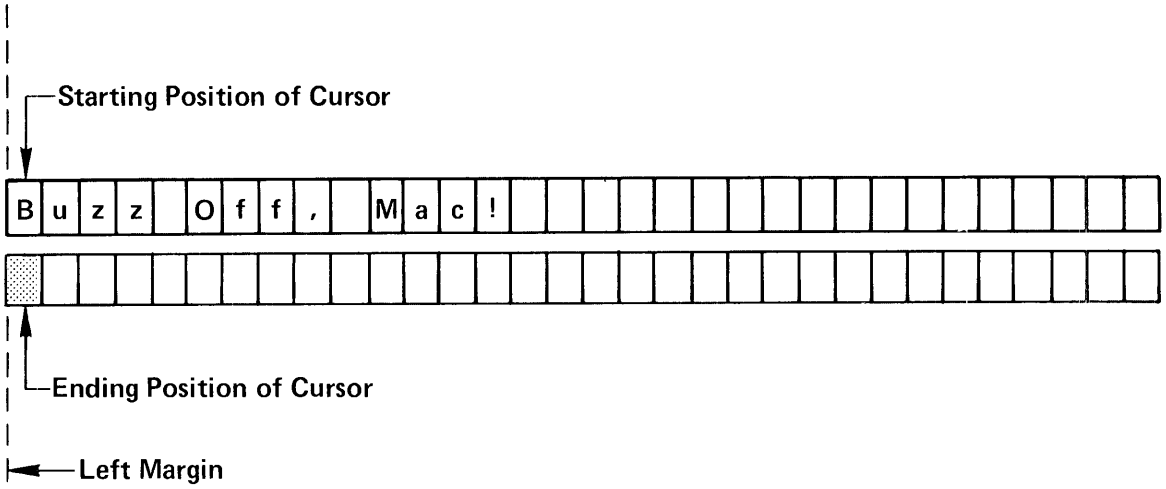
Printing a String Constant

If an alphanumeric character string is specified as a data item, then the string must be enclosed in quotation marks. The quotation marks are delimiters (separators) and are not printed.

Example 11—Printing a String Constant

```
220 PRINT "Buzz Off, Mac!"
```

GS Display Output



Notice in this example that a space is not inserted in front of the character string like it is with a numeric constant. Also notice that the characters inside the quotation marks are printed exactly as they are specified, both upper and lower case.

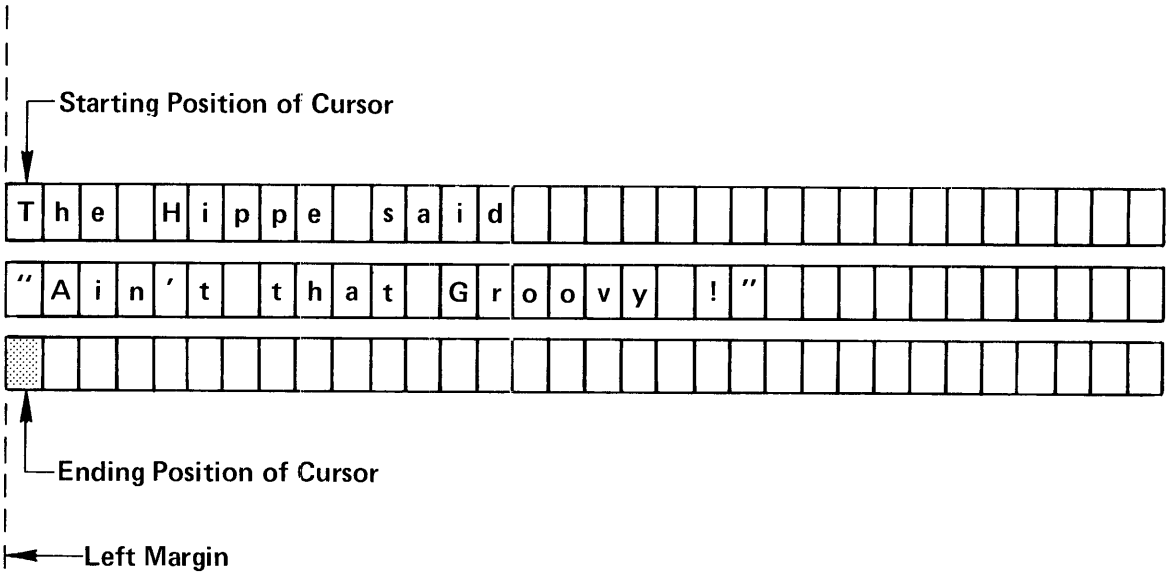
Printing String Variables

If a string variable is specified as a data item, then the string constant assigned to that variable is printed. If the specified string variable does not have an assigned value, then an error occurs and program execution is aborted.

Example 12—Printing a String Variable

```
230 LET B$ = "The Hippe said"
240 LET C$ = """Ain't that Groovy!"""
250 PRINT B$
260 PRINT C$
```

GS Display Output



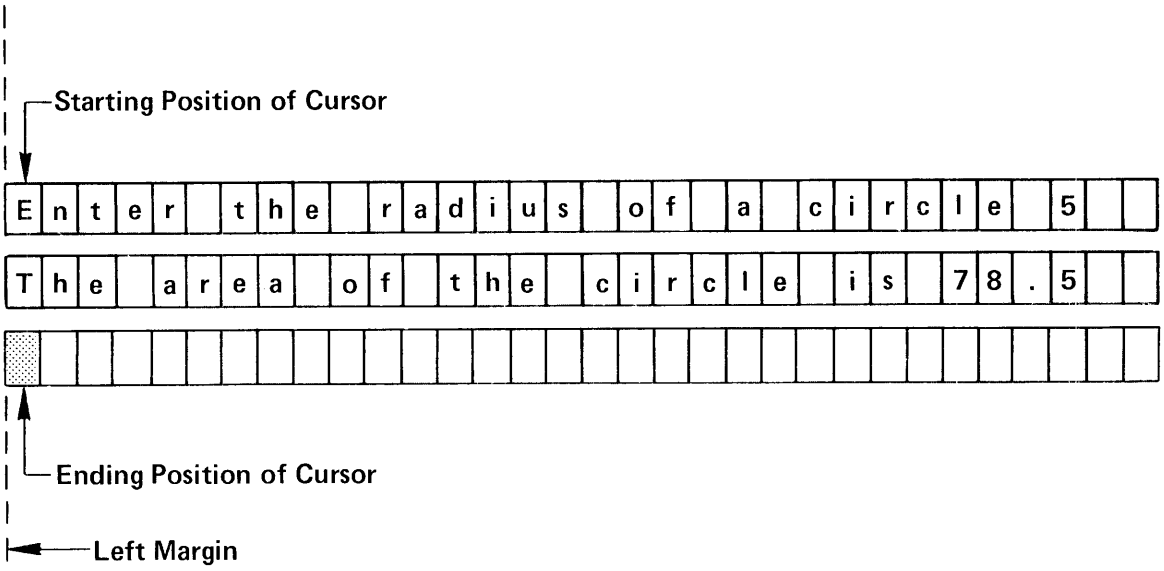
Printing Strings and Numeric Expressions in the Same PRINT Statement

Many times memory space can be saved by combining a printed message and a numeric expression in the same PRINT statement.

Example 13—Printing a String and a Numeric Expression in the Same Print Statement

```
270 PRINT "Enter the radius of a circle";
280 INPUT R
290 PRINT "The area of the circle is ";PI*R↑2
300 END
```

GS Display Output



In this example, line 270 asks the GS keyboard operator to enter the radius of a circle. Line 280 places the blinking question mark on the screen. (Assume 5 is entered from the keyboard.) The value 5 is assigned to the variable R in line 280 and program execution continues to line 290. In line 290, the message "The area of the circle is" is printed on the display. The numeric expression PI*R↑2 is then evaluated and the result (78.5) is printed on the GS display to the right of the printed message. Notice that the automatic insertion of a space before a numeric constant is suppressed if the numeric constant follows a string constant.

Intermixing Graphics and Alphanumerics

The alphanumeric cursor follows a predefined path on the screen to display printed lines of text. If graphic statements like MOVE and DRAW precede a PRINT statement, then the printed message may not line up with the normal path of the cursor. To restore the cursor to its normal path, a Carriage Return must be executed at the end of the PRINT statement; the RETURN key must be pressed; or a PAGE or HOME statement must be executed.

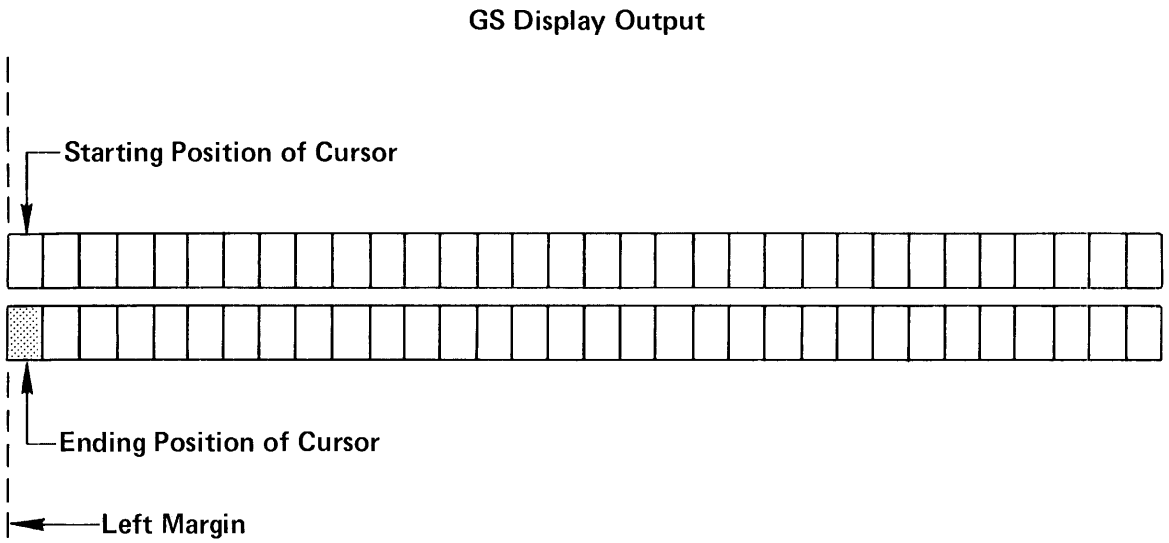
Printing ASCII Control Characters

There are seven display control functions which can be executed by sending ASCII control characters directly to the GS display. These control functions are BELL, BACKSPACE, HORIZONTAL TAB, LINE FEED, VERTICAL TAB, PAGE (or FORM FEED), and HOME. Control characters are entered into a BASIC statement by holding down the CTRL key on the GS keyboard and at the same time pressing the appropriate letter symbol. The control characters which affect the GS display are represented by the following letter symbols:

Control Character	Keyboard Input	Displayed Character	Function Performed
BEL (BELL)	CTRL G	G	Rings bell
BS (Backspace)	CTRL H	H	Backspaces the cursor
HT (Horizontal tab)	CTRL I	I	Tabs cursor to next tab stop
LF (Linefeed)	CTRL J	J	Moves cursor down one line
VT (Vertical tab)	CTRL K	K	Moves cursor up one line
FF (Form feed)	CTRL L	L	Erases screen and moves cursor up to Home
CR (Carriage Return)	CTRL M	Does not display character	Performs same function as RETURN key
RS (Record Separator)	CTRL ↑	↑	Returns the cursor to the HOME position

Example 14—Ringing the Graphic System Bell

```
300 PRINT "GGGGG"
```

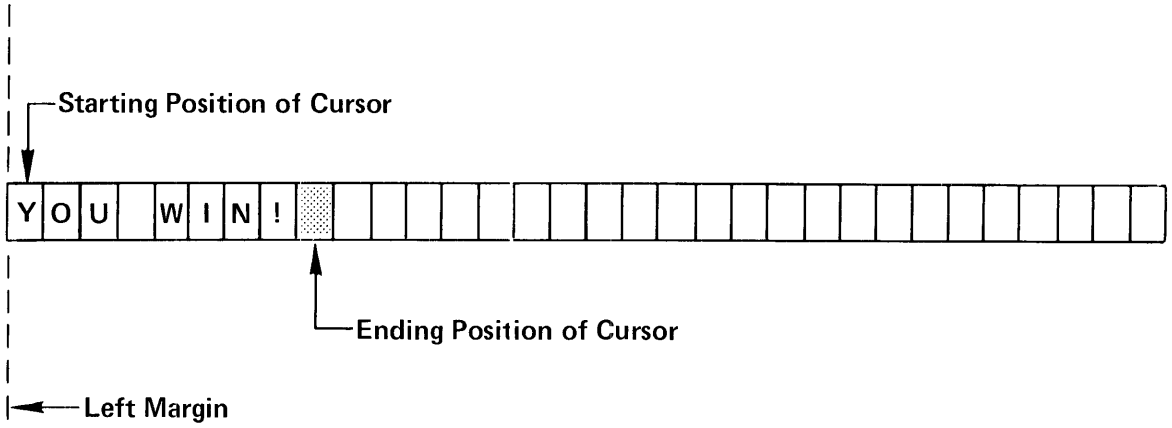


When line 300 is entered into memory from the GS keyboard, the CTRL G character is represented by an underlined G on the screen. When the statement is executed, however, the actual control character is sent to the display. This causes the Graphic System Bell to ring one time for each CTRL G character. In this case, the bell rings five times in succession when the string "GGGGG" is sent to the display. The cursor doesn't move when the CTRL G is executed; however, the carriage return/line feed (which automatically comes at the end of every PRINT statement) returns the cursor to the left margin. In this example, the cursor remains in its starting position until the bell rings five times, then moves down one line. To keep the cursor in its present position, a semicolon must be specified at the end of the PRINT statement.

Example 15—Mixing CTRL G characters in with a Character String

```
310 PRINT "YOU WINGGGGGGG!";
```

GS Display Output

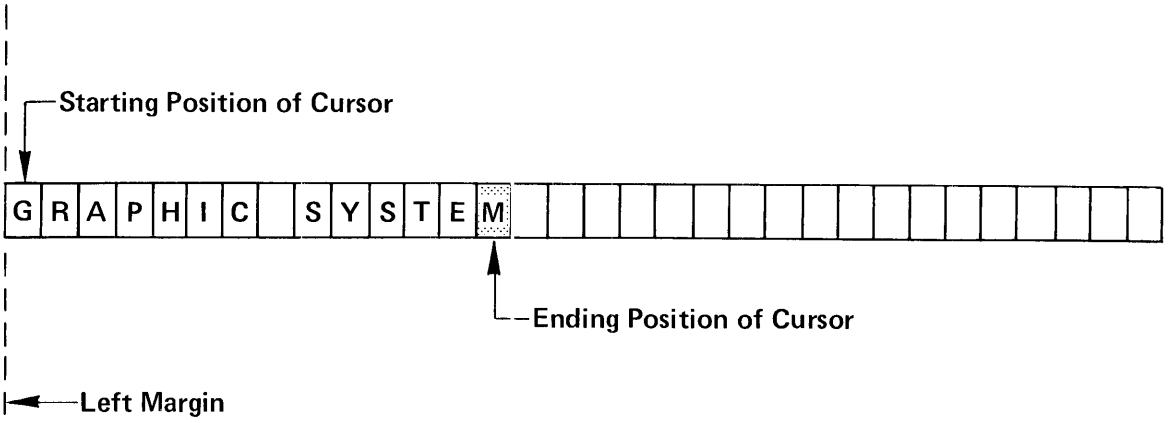


In this example, the characters YOU WIN are printed on the display when line 310 is executed under program control. The cursor finishes just beyond the N and waits there until the bell rings seven times. After the seventh ring, the exclamation mark (!) is printed and the cursor remains in the next position to the right because the automatic Carriage Return is suppressed with the semicolon at the end of the PRINT statement.

Example 16—Executing a BACKSPACE with the PRINT Statement

```
320 PRINT "GRAPHIC SYSTEMH";
```

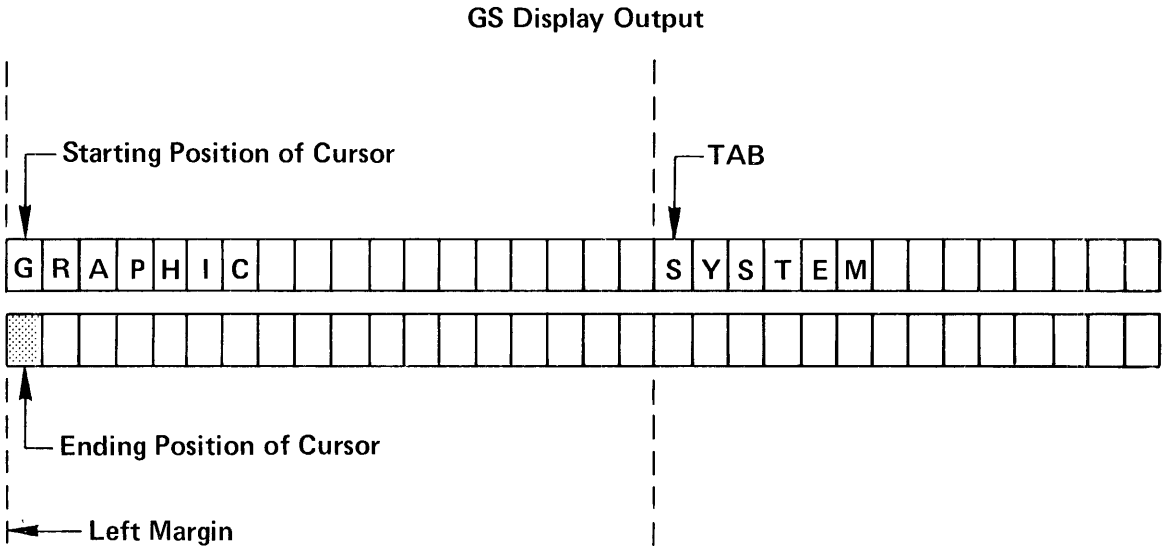
GS Display Output



When line 32Ø is executed, the character string "GRAPHIC SYSTEMH" is printed on the display starting at the present position of the cursor. The cursor finishes just beyond the M in "SYSTEM" then backs up one character position when the CTRL H is executed. The cursor remains over the M because the Carriage Return is suppressed by the semicolon at the end of the PRINT statement.

Example 17—Executing a Horizontal Tab with a PRINT Statement

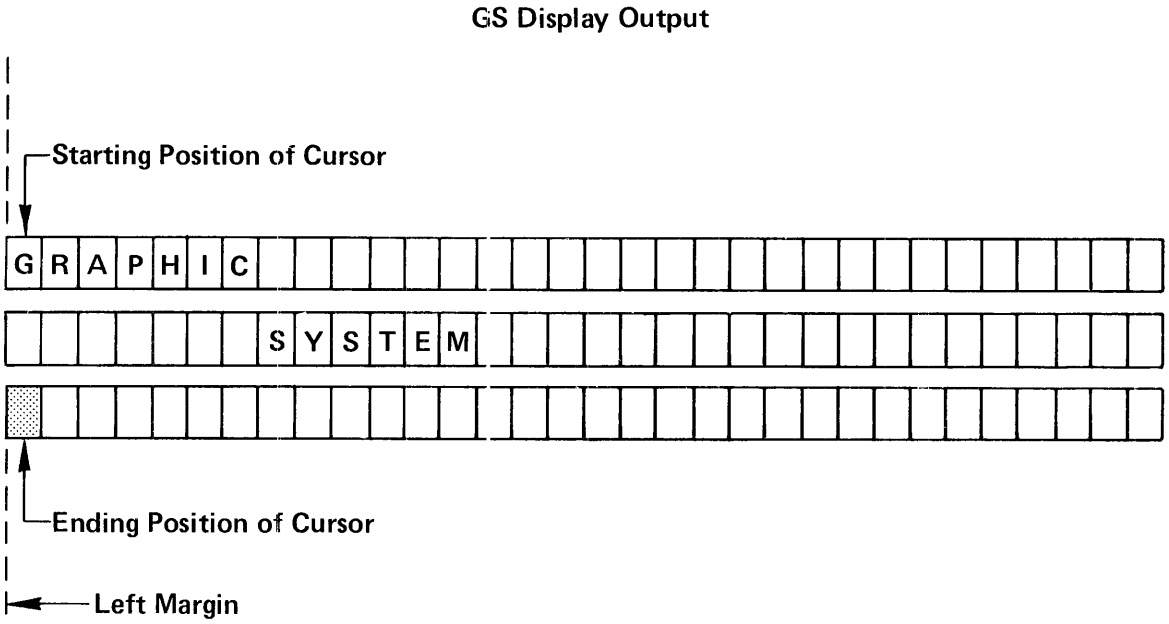
```
33Ø PRINT "GRAPHIC SYSTEM"
```



CTRL I causes the alphanumeric cursor to move to the next TAB position to the right. The TABs are preset to character positions 1, 19, 37, 55, and on larger screens, 73, 91, 109, and 127 in the default PRINT format. If the cursor is beyond the last TAB position when CTRL I is executed, then the cursor returns to the left margin and moves down one line. In the example above, the word GRAPHIC is printed starting at the present position of the cursor. The CTRL I character moves the cursor over to the next preset TAB position (position 19) and the word SYSTEM is printed. The automatic CR/LF returns the cursor to the left margin.

Example 18—Executing a Line Feed with the PRINT Statement

```
340 PRINT "GRAPHIC↓SYSTEM"
```

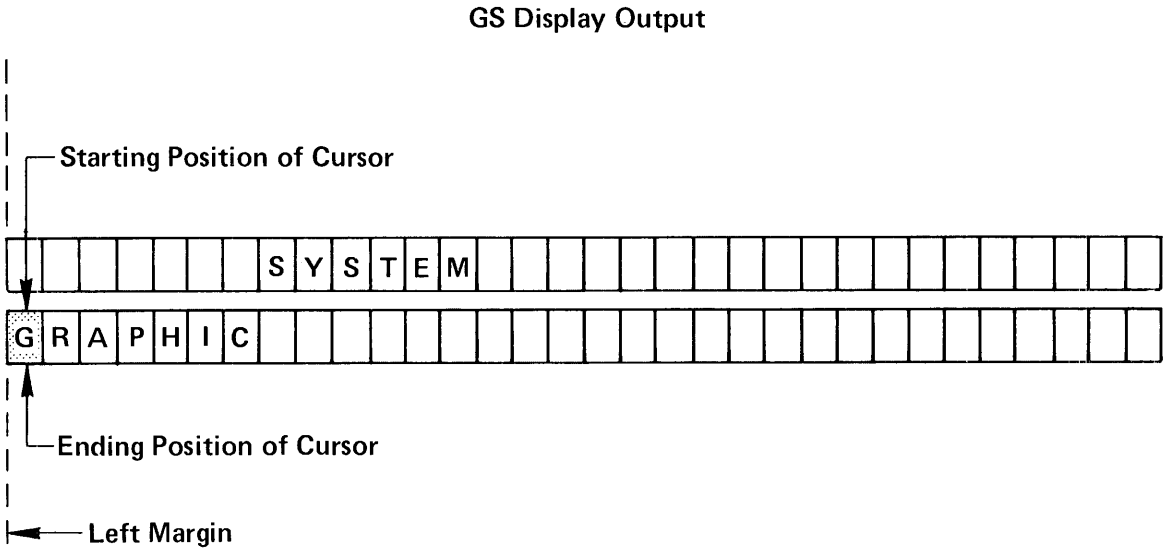


CTRL J moves the alphanumeric cursor down one character position. If the cursor is on line 35 when the CTRL J is executed, then a "page full" condition occurs and a blinking "F" appears in the upper left-hand corner of the screen. If this happens, the PAGE/HOME key must be pressed or the PAGE or HOME statement executed under program to return the cursor to the HOME position. (This is true only if the PAGE FULL environmental parameter is set to it's default value. Refer to PAGE FULL in the Environmental Control section for details.)

When line 340 is executed, the character string "GRAPHIC↓SYSTEM" is sent to the GS display. The word GRAPHIC is printed starting at the present position of the cursor. The CTRL J then moves the cursor down one line and the word SYSTEM is printed. The automatic Carriage Return at the end of the statement returns the cursor to the left margin.

Example 19—Executing a Vertical Tab with the PRINT Statement

```
350 PRINT "GRAPHICKSYSTEM"
```



CTRL K moves the alphanumeric cursor up one character position from its present location. If the cursor is on line 1 when CTRL K is executed, then wrap-around occurs and the cursor moves to the same horizontal position on line 35 (bottom of the screen).

When line 350 is executed, the character string "GRAPHICKSYSTEM" is sent to the GS display to be printed. The word GRAPHIC is printed, then the cursor is moved up one character position by the CTRL K; the word SYSTEM is then printed. Because a semicolon is not specified at the end of this PRINT statement, the automatic CR/LF moves the cursor to the left margin and down one space—to the original starting position.

Printing a Form Feed (CTRL L)

Sending a CTRL L character to the GS display is the same as executing a PAGE statement or pressing the PAGE key. The screen is erased and the alphanumeric cursor returns to the HOME position. For example:

```
360 PRINT "L HOWDY FRIEND!"
```

When line 36Ø is executed, the screen is erased and the cursor returns to the HOME position. The message "HOWDY FRIEND!" is printed on the screen. This method of executing a PAGE eliminates the need for a PAGE statement in the BASIC program.

Printing a Record Separator Character (CTRL ↑)

Sending a CTRL ↑ character to the GS display is the same as executing a HOME statement or pressing the HOME key on the GS keyboard. The alphanumeric cursor returns to the upper left-hand corner of the display. For example:

```
355 PRINT "↑";
```

When this statement is executed, the cursor returns to the home position; the screen is not erased.

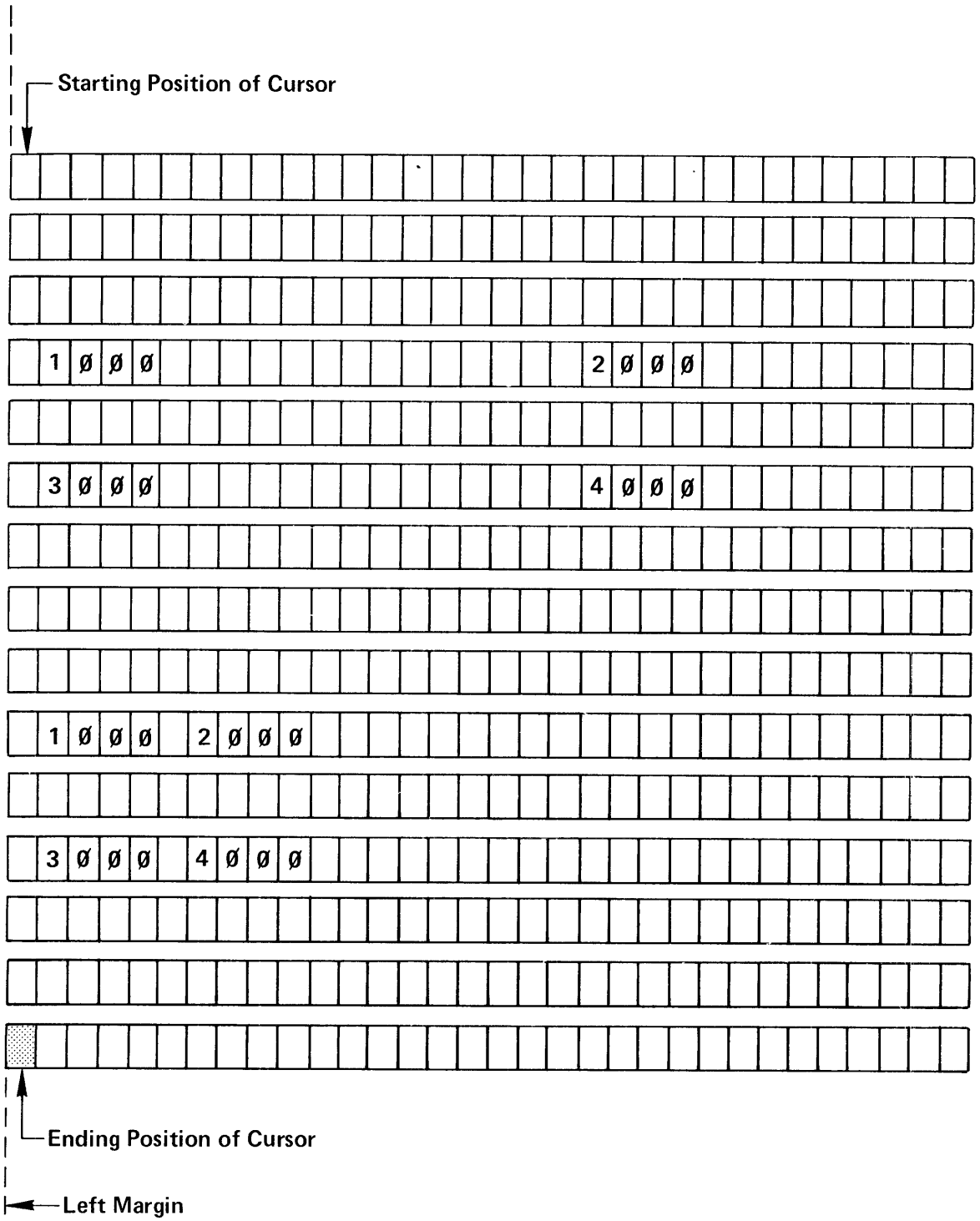
Printing Arrays

If an array variable is specified in a PRINT statement, the elements of array are sent to the GS display or to the specified peripheral device in row major order. If a semicolon is not specified after the array variable, then spaces are added so that each element fills an 18 character field. The elements are printed on the GS display row by row with one line separating each row. If a semicolon is specified after the array variable, the elements in each row are separated by one space. Vertically, the rows are separated by one line. The example below illustrates two array printouts; one with a semicolon specified after the array variable and one without a semicolon specified after the array variable.

Example 2Ø—Printing an Array

```
38Ø DIM A(2,2)
39Ø A(1,1)=1ØØØ
4ØØ A(1,2)=2ØØØ
41Ø A(2,1)=3ØØØ
42Ø A(2,2)=4ØØØ
43Ø PRINT A,A;
```

GS Display Output



When statement 43Ø is executed, the array A is printed twice, as shown in the illustration; the first time with the TAB spacing, the second time without the TAB spacing. Notice that the cursor always moves down three lines before and after the array is printed. This is done to improve the appearance of the printout.

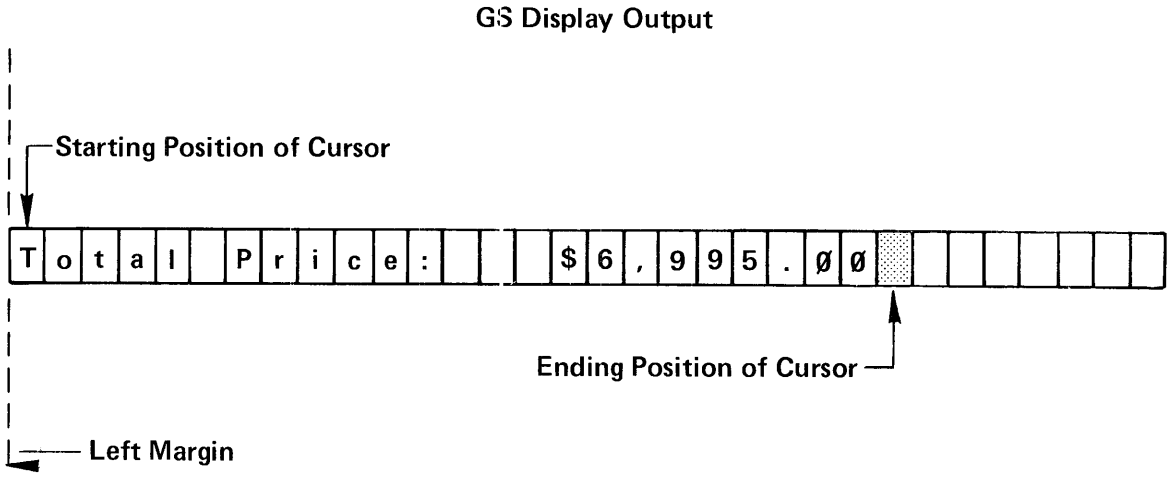
Specifying a Print Format

If a "format string" is specified in a PRINT statement, the specified data items are converted into an ASCII character string using the "format string" as a guide. By using a format string as a guide, the printed output can be shaped into any predefined format. For example, special print fields can be set up so that a dollar sign is placed in front of each numeric value. In addition, commas can be added to the numbers to indicate thousand dollar divisions, a TAB to any character position can be specified, spaces added, and special characters inserted to give an almost infinite variety of output.

A format string can be specified in a PRINT statement in three ways. Each example which follows illustrates a different method to specify a format string.

Example 21—Specifying a Format String Directly in a PRINT Statement

```
36Ø LET M5 = 6995
37Ø PRINT USING """Total Price:""3X$CFD.2DS":M5
```

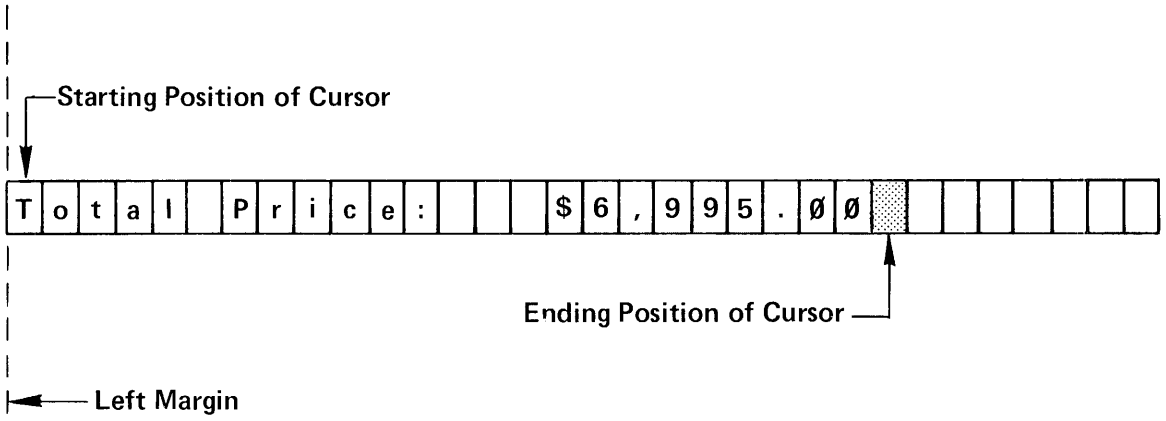


In this example, the format string """"Total Price:""""3X$CFD.2DS" is specified directly in the PRINT statement. The assigned value of the variable M5 (6995) is printed on the GS display using the format string as a guide. The results are shown in the illustration. (Refer to the explanation of the IMAGE statement for a guide to the symbology used in the format string.)

Example 22—Using a String Variable to Specify a Format String

```
38Ø A$ = """"Total Price:""""3X$CFD.2DS"
39Ø M5 = 6995
4ØØ PRINT USING A$ :M5
```

GS Display Output

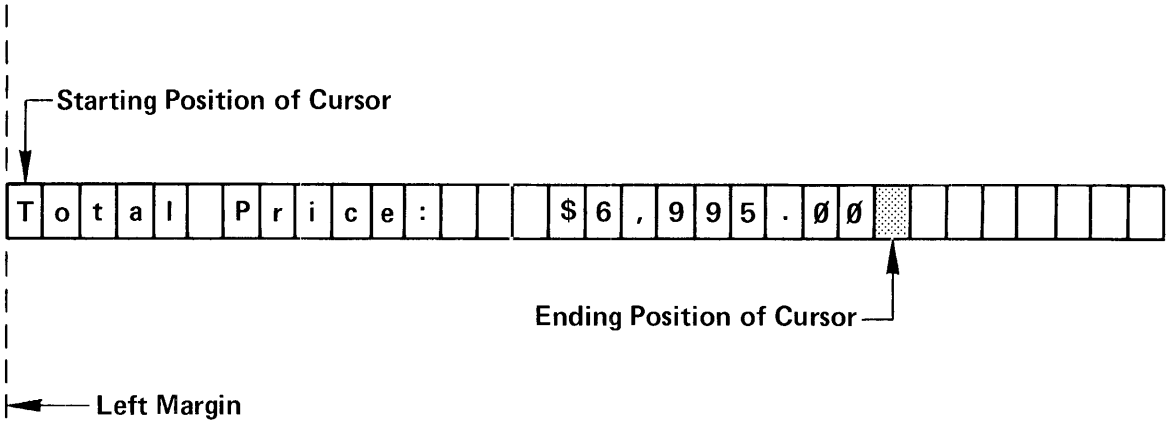


In line 38Ø, the format string """"Total Price:""""3X$CFD.2D" is assigned to the string variable A$ and the number 6995 is assigned to the numeric variable M5 in line 39Ø. In line 4ØØ, the value of M5 is printed on the GS display using the format specified by A$. The output is the same as the previous example, however, this method of specifying the format is much easier. In the first place, the PRINT USING statement is not as cluttered with information. Secondly, if the print format is used again later in the program, it is easier to specify A$ rather than retype the entire format string into the PRINT statement.

Example 23—Specifying a Format String in an IMAGE Statement

```
41Ø IMAGE "Total Price:"3X$CFD.2DS
42Ø M5 = 6995
43Ø PRINT USING 41Ø: M5
```

GS Display Output



This example shows the third format string specification method. In this case, the format string is specified in line 41Ø, an IMAGE statement. Notice in line 43Ø that the format to be used is specified by referring to the IMAGE statement number. This method has an advantage over the method just described in that a string variable is not used up to specify the format. Since there are only 26 possible string variables (A$-Z$), this method has great advantage in a program which makes heavy use of string variables. Refer now to the explanation of the IMAGE statement for complete description of the symbology used in format strings.

Sending ASCII Data to the Internal Magnetic Tape Unit

Creating Logical Records

The PRINT statement is used to send ASCII data to the internal magnetic unit. This is accomplished by specifying the primary address @33: after the keyword PRINT. The data items to be sent are listed after the primary address. For example:

```
1ØØ PRINT @33:468590;"Silver,I.O. ";2Ø;"Winter 1975 ";1.86
```

When this statement is executed, the BASIC interpreter converts the entire list of data items into an ASCII character string using the default print format. Once the data list is converted, a distinction between the character strings and the numeric constants cannot be made. In this case, the entire data list is sent as a 37 character ASCII data string with a Carriage Return tacked onto the end. This data string is treated as a logical record on magnetic tape. In this example, the data string represents a student record. The first numeric value is the student number; the student name comes next; the numeric value 2Ø is the number of credit hours for the term; the name of the term (Winter 1975) follows; and the final numeric value is the student's grade point for the term. Since the difference between numeric data and string data cannot be

determined after the data is converted into ASCII format, the entire string is treated as one unit; a logical record. When the information is brought back into memory from the magnetic tape with the INPUT statement, the entire logical record is treated as a unit and assigned to a specified string variable. (Refer to INPUT in this section for complete information on inputting logical records from ASCII data files.) Notice that semicolons are used to separate data items in the PRINT statement. This eliminates the addition of needless blanks which are added if commas are used.

Sending ASCII Data to a New File

Before ASCII data is sent to a new magnetic tape file, the file must be created and the magnetic tape read/write head must be positioned to the beginning of the file. The following program illustrates how to create a file on tape, find the beginning of the file, then send a logical record to the file.

```
1ØØØ FIND Ø
1Ø1Ø MARK 1,1ØØØ
1Ø2Ø FIND 1
1Ø3Ø PRINT @33:468590;Silver,I.O. ";2Ø;"Winter 1975 ";1.86
1Ø4Ø FIND 1
```

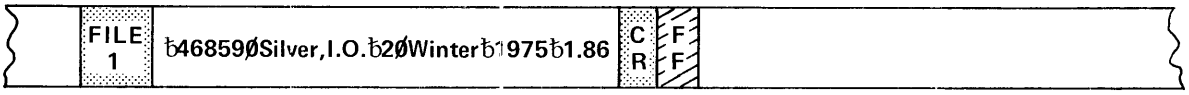
Line 1ØØØ in this program rewinds the tape to the load point sensor (the beginning). Line 1Ø1Ø creates a new file on the tape which is 1ØØØ bytes long (1Ø24 to be exact). Line 1Ø2Ø positions the read/write head to the beginning of the file. Line 1Ø3Ø then sends the specified logical record to the new file as an ASCII character string. This record is recorded starting at the present position of the read/write head.

The last two characters printed are the Carriage Return character to mark the end of the logical record and the End of File character (hexidecimal FF) which marks the logical end of the file. (The term "logical end of the file" refers to the placement of the EOF character after the last logical record in the file. The logical length of the file may be considerably shorter than the physical length and grows as more logical records are added to the file.)

Line 1Ø4Ø closes the file. This is a very important step in sending ASCII data to the magnetic tape. Normally, the internal tape unit holds the data in a 256 byte memory buffer until the buffer is full. The 256 bytes of data are then "dumped" into the ASCII data file all at once. In this case, the logical record is held in the buffer because it doesn't contain 256 bytes. If power is removed from the system at this point, then the record never reaches the tape and is lost forever. Line 1Ø4Ø forces the magnetic tape buffer to "dump" its contents into the file. The read/write head is then repositioned to the beginning of the file. This is called "closing the file." (Closing the file

can also be done by executing a CLOSE statement or an END statement.) Because the first operation on the file is a PRINT operation, the file header is marked "ASCII DATA." This is done just before the ASCII data is stored in the file.

The following illustration shows the results of the above operation.



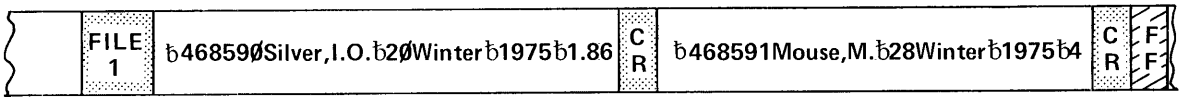
Magnetic Tape

Adding Logical Records to a Partially Filled Data File

Logical records can be added to a partially filled data file, but first the logical end of the file must be found. The following program shows how to position the magnetic tape read/write head to the logical end of an ASCII data file, then add logical records to the file.

```
105Ø ON EOF (Ø)THEN 109Ø
106Ø FIND 1
107Ø INPUT @33:A$
108Ø GOTO 107Ø
109Ø PRINT @33:468591;"Mouse,M. ";28;"Winter 1975 ";4
11ØØ CLOSE
```

This program activates the End Of File ON Unit in line 1Ø5Ø. This tells the BASIC interpreter to be on the lookout for an End Of File condition and to transfer program execution to line 1Ø9Ø when it occurs. (Refer to the ON. . .THEN. . .statement in the Handling Interrupts section for details.) Line 1Ø6Ø locates the beginning of file 1—the ASCII data file just created. Because ASCII data files must be read sequentially, it is necessary to start at the beginning of the file and read each record until the EOF character is found. Lines 1Ø7Ø and 1Ø8Ø accomplish this task. When line 1Ø7Ø is executed, the first record in the file is input into memory and assigned to the string variable A$. Line 1Ø8Ø then tells the BASIC interpreter to go back to line 1Ø7Ø and do it again. The next logical record is input into memory and is assigned to A$. This, of course, overwrites the first record; but it's all right in this case because the program is only trying to locate the End Of File character; the information assigned to A$ isn't used. Line 1Ø7Ø and 1Ø8Ø causes the magnetic tape read/write head to advance through the file until the logical end of the file is found. When an attempt is made to input the EOF character, the EOF ON UNIT activated in line 1Ø5Ø alerts the BASIC interpreter. Program control is transferred immediately to line 1Ø9Ø where a new logical record is added to the end of the file. Because the magnetic tape read/write head is positioned on the EOF character when line 1Ø9Ø is executed, the new logical record overwrites the old EOF character. A new EOF character is placed at the end of the new logical record to mark the new logical end of the file. The illustration below shows the results of the above operation.



Magnetic Tape

Sending ASCII Data to a Binary Data File or a Program File

An old binary data file or an old program file can be reused to store ASCII data if necessary. The procedure is the same as storing ASCII data in a new file. Position the tape head to the beginning of the old file with the FIND statement and then execute a PRINT statement to store the data. The old information in the file is overwritten with the new ASCII data. Any information beyond the ASCII data cannot be accessed because an EOF character is placed after the last logical record from the PRINT statement. In other words, the information in the old file before the PRINT operation is lost forever. The old file header is remarked "ASCII DATA" for TLIST operations. Executing a KILL statement on the old file is a necessary first step if the file contains old information stored in binary format.

Sending ASCII Data to an External Peripheral Device

The PRINT statement is also used to send ASCII data to an external peripheral device connected to the General Purpose Interface Bus (GPIB). To do so, the appropriate primary address must be specified after the keyword PRINT. The data items are listed after the primary address. For example:

```
1110 PRINT @9: 468592;"Rabbit,P.";20;"Winter 1975";3.35
```

When this line is executed, the BASIC interpreter issues the I/O address @9,12: over the GPIB. Primary address 9 tells peripheral device number 9 that it has been selected to take part in the upcoming I/O operation. Secondary address 12 is issued by default and tells device number 9 that the BASIC interpreter is executing a PRINT statement and to prepare to receive an ASCII character string. The BASIC interpreter then sends the specified data items to the peripheral device as an ASCII character string over the GPIB.

Refreshing a Character On the GS Display

The GS Display has the facility to print a character without storing the character on the screen. The I/O address @32,24: must be specified to exercise this facility. For example:

```
100 PRINT @32,24:"*"
```

When this statement is executed, the character "*" is printed without being stored.

If the PRINT statement is continually executed, the specified character symbol appears on the screen for a longer period of time without storing. For example:

```
100 INIT
110 PAGE
120 MOVE 65,50
130 FOR I=1 TO 40
140 PRINT @32,24:"*"
150 NEXT I
160 END
```

This program moves the display cursor to the center of the screen and prints the refresh character "*" 40 times (on the 4051 Graphic System, for about 10 seconds).

Small routines like this can be inserted into programs to enhance graphic output and make it come alive. This technique is used in the first frame of the Graphic System tutorial to move the "o" character from the right side of the screen to a position over the "i" in the word tutorial. The "o" is then stored as the dot over the "i".

THE RBYTE STATEMENT

Syntax Form:

[Line number] RBY numeric variable [, numeric variable] ...

Descriptive Form:

[Line number] RBYTE target variable for incoming data byte [, target variable for incoming data byte] ...

Purpose

The RBYTE (Read Byte) statement receives one or more data bytes from a peripheral device on the General Purpose Interface Bus (GPIB) and assigns each data byte to a numeric variable.

Explanation

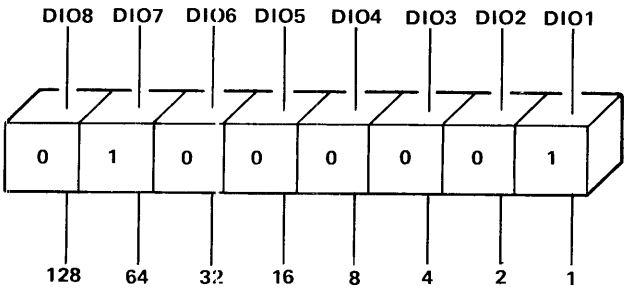
Reviewing the GPIB Hardware Features

If you're unfamiliar with the General Purpose Interface Bus operation, at this point it is appropriate to review the hardware features of the GPIB which are found in Appendix C.

Binary-to-Decimal Conversion

There can only be 256 unique 8-bit patterns transferred over the GPIB because the Data Bus is only 8 lines wide. In the WBYTE and RBYTE statements, each bit pattern (byte) is represented by a decimal number within the range Ø through 255. If the decimal number is preceded by a minus sign, then the EOI (End or Identify) signal line on the GPIB Management Bus is activated when the byte is transferred.

Data bytes received by the BASIC interpreter over the GPIB are converted to their decimal equivalents before they are assigned to numeric variables. The following illustration shows how an eight-bit binary code is converted to a decimal number.



64 + 1 = 65

Each line on the Data Bus represents a binary bit. If the line is electrically high (near +5 Vdc), the line represents a binary Ø. If the line is electrically low (near Ø Vdc), the line represents a binary 1. In the illustration above, DIO1 and DIO7 are electrically low, so they represent a binary 1; the rest of the lines are high; they represent a binary Ø.

Each bit in the eight-bit code carries a decimal weight as shown under each block in the illustration. The right most bit (DIO1) carries a weight of 1; the seventh bit (DIO7) carries a weight of 64. Adding these weights together gives the decimal equivalent of the byte. In this case, the byte is equivalent to 65 (base 1Ø).

Receiving Data Bytes from an External Peripheral Device

The following program illustrates how to assign an external peripheral device as a talker and receive a data byte over the GPIB.

```
1ØØ INIT
11Ø WBYTE @8Ø,1Ø8:
12Ø RBYTE X
13Ø WBYTE @63,95:
14Ø M$=CHR(X)
15Ø PRINT M$
16Ø END
```

When line 1ØØ is executed, the system environmental parameters are initialized and IFC (Interface Clear) on the GPIB is activated. This places the interface circuitry for each peripheral device on the GPIB in a known quiescent state. Line 11Ø assigns peripheral device number 16 to be a talker. The @ symbol activates the ATN (Attention) signal line on the GPIB. This tells each peripheral device on the bus that the data bytes to follow represent peripheral addresses or control instructions from the controller. The primary talk address for peripheral device 16 (decimal 8Ø) is sent over the bus first. This tells device 16 that it has been selected as the talker in the upcoming data transfer and to start talking as soon as ATN is released. A secondary address (decimal 1Ø8) immediately follows and has a predefined meaning to device 16. Assume in this case that secondary address 1Ø8 tells device 16 to start sending data bytes over the Data Bus starting at the present position of its write head. (Refer to the WBYTE statement in this section or Appendix A for a complete list of GPIB primary and secondary addresses.)

After the primary and secondary addresses for device 16 are issued, the colon in the WBYTE statement causes the controller to release the ATN signal line. This tells each peripheral device on the bus that only those devices addressed while ATN was active can take part in the data transfer; in this case, only device 16 can participate.

NOTE

Whenever a receive data command such as RBYTE is executed, the Graphic System, acting as the controller, assigns itself as a listener even though it is not explicitly specified as a listener by appropriate bus protocol.

At this point in the operation, device 16 takes over the bus as the talker in charge. Device 16 waits for the listener to release NRFD (Not Ready For Data), then places an eight-bit binary code on the Data Bus. (Assume that the code is decimal 65 for illustrative purposes.) Device 16 activates the DAV (Data Valid) signal line to tell the listener that the data byte is ready for transfer. The listener (in this case, the Graphic System) captures the data byte from the bus, converts the 8-bit binary code to its decimal equivalent (65) and assigns the value to the numeric variable X. If device 16 activates the EOI signal line while the data byte is transferred, the decimal number is negated to −65 and assigned to the variable X. Since only one variable is specified in this RBYTE statement, program execution continues to line 13Ø after one byte is transferred. If more than one variable is specified in line 12Ø, then the BASIC interpreter keeps receiving data bytes from device 16 until each variable has an assigned value.

Line 13Ø activates the ATN signal line again and issues the universal commands UNTALK (decimal 63) and UNLISTEN (decimal 95) over the GPIB. This places all active peripheral devices in a known quiescent state and terminates the transfer.

The program at this point has the freedom to interpret the meaning of the data byte assigned to X in any way it sees fit. In this case, assume that the binary code represents an ASCII character. In line 14Ø, the CHR function is used to convert the decimal number to its ASCII character equivalent. (Refer to the CHR function in the Character String section and the ASCII Character Value Chart in Appendix B for further explanation of the CHR function.) The ASCII letter "A" is equivalent to decimal 65, so "A" is assigned to the string variable M$ in line 14Ø. In line 15Ø, "A" is printed on the GS display for viewing.

This program illustrates the basic principles of byte transfer over the GPIB using WBYTE and RBYTE. Elaborate programs can be written to exercise the full capabilities of the interface. With a little imagination and a little practice, you'll find that the RBYTE statement and the WBYTE statement provide a primitive, yet almost unlimited capability to interface the Graphic System to the outside world.

THE READ STATEMENT

Syntax Form:

[Line number] REA [I/O address] { array variable / string variable / numeric variable } [, { array variable / string variable / numeric variable }] ...

Descriptive Form:

[Line number] READ [I/O address] target variables for incoming data

items formatted in machine dependent binary code

Purpose

The READ statement brings in data items from the specified peripheral device and assigns those items to the specified variables. If a peripheral device is not specified, the DATA statement in the current BASIC program is selected as the data item source. The incoming data items must be formatted in machine dependent binary code.

Explanation

Reading Data Stored in the DATA Statement.

If an I/O address is not specified in a READ statement, then data items are assigned to the specified variables from the DATA statement. If the current BASIC program doesn't have a DATA statement, then an error occurs and program execution is aborted.

The following program illustrates how data items can be arranged in a DATA statement and how those items are assigned to variables with the READ statement:

```
100 INIT
110 DATA "Sally",5,"1305 S.W. Henry St."
120 DATA 6.95, "Billy",6,"1501 S.E. Morrison",5.87
130 FOR I=1 TO 2
140 READ A$,B,C$,D
150 PRINT USING 180: A$,B
160 PRINT USING 190: C$,D
170 NEXT I
```

```
18Ø IMAGE "STUDENT NAME:",2X,1ØA,/,"AGE:",2X,FD
19Ø IMAGE "ADDRESS:",2X,3ØA,/,"ART SUPPLIES:",2X,$+FD.FD,3/
2ØØ END
```

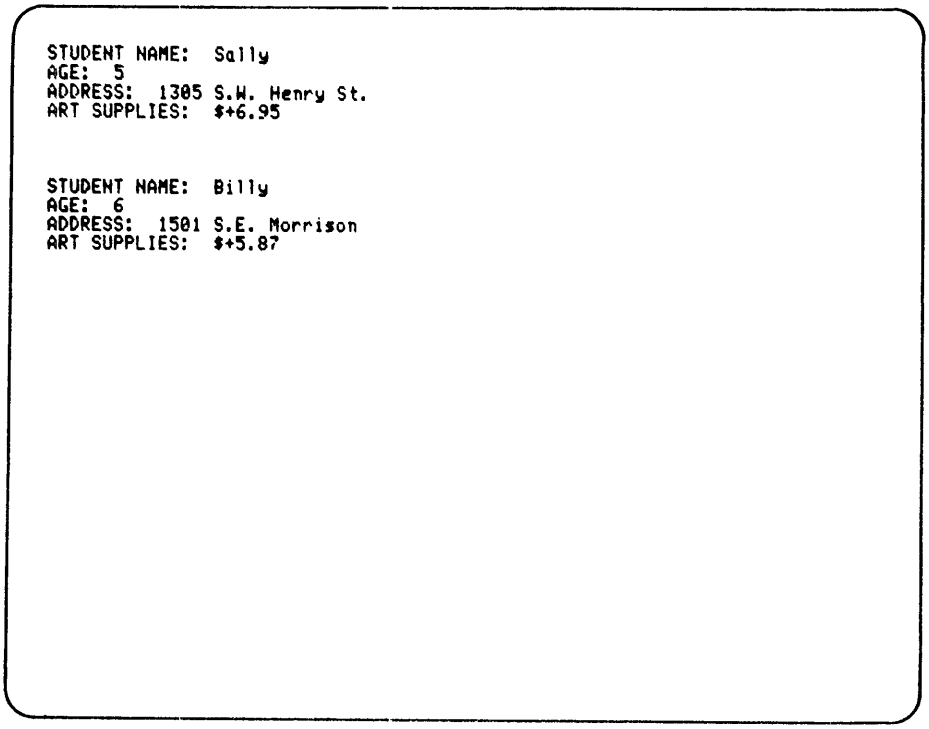
When line 1ØØ in this program is executed, the system parameters are set to their default values and the DATA statement pointer is set to the first data item in the first DATA statement; in this case, the pointer is set to the string constant "Sally" in line 11Ø.

Lines 13Ø through 17Ø read the data items from the DATA statement and print the items on the GS display. The operation is executed as follows. The first time line 14Ø is executed, the string constant "Sally" is assigned to the variable A$ and the DATA statement pointer moves to the next data item (5). The 5 is assigned to the variable B and the DATA statement pointer moves to the string constant "13Ø5 S.W. Henry St.". This string constant is assigned to the variable C$. Because the end of this DATA statement is reached here, the DATA statement pointer automatically moves to the next DATA statement and points to the first data item in that statement; in this case, the pointer moves to numeric constant 6.95. This value is assigned to the variable D and the DATA statement pointer moves to the string constant "Billy." Since the variable D is the last variable specified in the READ statement, the read operation is finished and the DATA statement pointer remains pointing to "Billy"; this, of course, indicates that "Billy" is the next data item to be read.

When lines 15Ø and 16Ø are executed, the data items assigned to the variables A$,B,C$, and D are printed on the GS display according to the print format specified in lines 18Ø and 19Ø. (Refer to the IMAGE statement in this section for details on print formats.)

Line 17Ø returns program control to line 14Ø, the READ statement, and the read operation is re-executed. This time the DATA statement pointer is pointing to the string constant "Billy," so "Billy" is assigned to A$. (This, of course, overwrites the previous value of A$ which was "Sally".) The numeric constant 6 is assigned to the variable B, the string constant "15Ø1 S.E. Morrison" is assigned to the variable C$, the numeric constant 5.87 is assigned to the variable D. This ends the second READ operation and the DATA statement point is left pointing out into space. This happens because the numeric constant 5.87 is the last data item in the last DATA statement. At this point, the DATA statement point must be reset with a RESTORE statement or an INIT statement before another READ operation is executed. If not, a read error occurs and program execution is aborted. In this program, however, a RESTORE statement is not necessary, because the data items just read are printed in lines 15Ø and 16Ø and program execution is terminated in line 2ØØ. The results are shown in the following illustration:

GS Display Output



Restoring the DATA Statement Pointer before Executing a Program

It is good programming practice to restore the DATA statement pointer with an INIT statement or a RESTORE statement before a program is executed. For example, RUN 1ØØ does not restore the pointer, and if a program is run more than once, then the DATA statement pointer might be pointing out to "right field" (or some obscure place). If the pointer does not point to a valid data item in the current program, then a read error occurs the first time a READ statement is executed in the program. Executing the RUN statement without parameters and the OLD statement does, however, restore the DATA statement pointer. (Refer to the RESTORE statement in this section for details.)

Reading Matrices

If a previously dimensioned variable is specified in a READ statement, then numeric data items from the data source are assigned to the elements of the array in row major order until each element has an assigned value. If the data source does not have enough numeric data items to fill the array, or if a character string is imbedded in between the numeric data items, then a read error occurs and program execution is aborted. For example:

```
10Ø INIT
11Ø DATA 1,2,3,4,5,6,7,8,9
12Ø DIM A(4),B(2,2)
13Ø READ A,B
14Ø PRINT A,B
15Ø END
```

When line 10Ø is executed, the system is initialized and the DATA statement pointer is restored to the first data item in line 11Ø. Line 11Ø defines the data items to be read. When line 12Ø is executed, the variable A is dimensioned as a one-dimensional array with four elements; the variable B is dimensioned as a two-dimensional array with two rows and two columns. When line 13Ø is executed, the following assignments are made:

```
A(1)=1          B(1,1)=5
A(2)=2          B(1,2)=6
A(3)=3          B(2,1)=7
A(4)=4          B(2,2)=8
```

The DATA statement pointer is left pointing to 9. When line 14Ø is executed, the arrays are printed on the GS display. The program ends in line 15Ø.

Reading Binary Data Files on the Internal Magnetic Tape

The READ statement must be used to retrieve data items stored on the internal magnetic tape with the WRITE statement. Specifying the I/O address @33: after the keyword READ selects the internal magnetic tape as the source for binary data. Before a READ operation involving the internal magnetic tape is executed, the tape head must be positioned to the beginning of a binary data file. For example:

```
10Ø  FIND 5
12Ø  READ @33: A,B,C$,D
```

When line 1ØØ is executed, the internal magnetic tape head is positioned to the beginning of the storage area in file 5. Line 12Ø then assigns the first data item to the variable A, the second data item to the variable B, the third data item to the variable C$, and the fourth data item to the variable D.

If the data in file 5 is stored in ASCII format, then an error occurs and program execution is aborted. If there is a data item mismatch, for example, if the first data item in the file is a character string instead of a numeric value, then an error occurs and program execution is aborted.

When the data item type is unknown, the TYP function can be used to find out. (See the explanation of the TYP function in this section for details.)

Reading Binary Data from an External Peripheral Device

Data in binary format can be transferred from an external peripheral device into memory and assigned to the specified variable, if the appropriate I/O address is specified in the READ statement. Normally, the data to be read is first sent to the peripheral device for storage with the WRITE statement. A data transfer in binary format is initiated with the READ statement as follows:

```
34Ø READ @28:M,R,W
```

When this statement is executed, the BASIC interpreter issues the I/O address @28,14: over the General Purpose Interface Bus (GPIB).

Primary address 28 tells peripheral device number 28 that it has been selected to take part in an I/O operation. Secondary address 14 is issued by default and tells device 28 to send data items in binary format starting at the present position of its read head.

Device 28 responds by sending data items over the GPIB. The first data item transferred is assigned to the variable M, the second data item is assigned to the variable R, and the third data item is assigned to the variable W. All three data items must be numeric values in this case, or an error occurs and program execution is aborted. After the third data item is transferred over the

GPIB, the BASIC interpreter issues the universal commands UNTALK and UNLISTEN to device 28. This terminates the transfer and places device 28 in a quiescent condition. (Refer to the WRITE statement in this section for details on the internal binary format used by Graphic System.)

Primary Address 34 Specifies the DATA Statement as the Data Source

The DATA statement is specified as the data source for an I/O operation by specifying primary address 34. Normally, primary address 34 is selected by default for READ operations, but if the primary address is specified as a variable in a READ statement, then the variable must be assigned the value 34 to select the DATA statement as the data source. For example:

```
265 READ @A:X,Y,Z$
```

When line 265 is executed under program control, the numeric constant assigned to A specifies the data source. If A equals 34, then the DATA statement is selected as the data source. If A changes to 33, then the current file on the internal magnetic tape unit is selected as the data source. And, if A changes to 15, for example, then device number 15 on the General Purpose Interface Bus (GPIB) is selected as the data source.

THE RESTORE STATEMENT

Syntax Form:

[Line number] RES [line number]

Descriptive Form:

[Line number] RESTORE [line number]

Purpose

The RESTORE statement positions the DATA statement pointer to the first data item in the specified DATA statement. If a DATA statement line number is not specified, the pointer is positioned to the first data item in the DATA statement with the lowest line number.

Explanation

The DATA Statement

The DATA statement acts like an internal data file to a BASIC program. If a program contains more than one DATA statement, the DATA statements are linked together in a continuous chain starting with the lowest line numbered DATA statement in memory and ending with highest line numbered DATA statement in memory.

DATA Statement Pointer

An invisible pointer is associated with the DATA statement to indicate which data item is to be read next. This pointer is set to the first data item in the first DATA statement on system power up, after the execution of an INIT statement, and after the execution of a RESTORE statement.

Reading Data from a DATA Statement

The READ statement is used to assign each data item in the DATA statement to a variable. The data item indicated by the DATA statement pointer is assigned to a variable specified in the READ statement; numeric data is assigned to numeric variables and string data are assigned to

string variables. After a data item is assigned to a variable, the DATA statement pointer moves one position to the right in the DATA statement. The next READ operation assigns this data item to the next variable. (Refer to the READ statement in this section for complete details.)

Reaching the End of a DATA Statement

When the last data item in a DATA statement is read, the pointer moves to the first data item in the next DATA statement. After the last data item in the last DATA statement is READ, the pointer moves off the end of the line and points out into space. If a READ operation is attempted with the pointer in this position, a read error occurs, and program execution is aborted.

Restoring the DATA Statement Pointer

The RESTORE statement restores the DATA statement pointer to the beginning of a DATA statement. For example, the statement . . .

```
120 RESTORE 500
```

restores the pointer to the first data item in line 500. Line 500 must be a DATA statement, in this case, or an error occurs and program execution is aborted.

If a line number is not specified in the RESTORE statement, then the pointer is restored to the first data item in the lowest numbered DATA statement. For example:

```
100 INIT
110 DATA 165,2.8E-045, "Mac",1000,"Frank"
120 READ A,B,C$,D
130 DATA 1,-.005,34
140 RESTORE
150 READ X
```

When this program is executed, line 120 reads the data in line 110. The number 165 is assigned to the variable A, the number 2.8E-045 is assigned to the variable B, the string constant "Mac" is assigned to the string variable C$, and the number 1000 is assigned to the variable D. The DATA statement pointer now points to "Frank," the last data item in line 110. When line 140 is executed, the DATA statement pointer is restored. Because a line number is not specified as a parameter, the pointer is positioned to the number 165, the first data item in line 100. When line 150 is executed, the number 165 is again assigned, only this time to the variable X. If line 140 were not in the program, then an attempt to assign "Frank" to the variable X would be made because "Frank" would be the next data item to be read. This would result in an error.

Restoring the DATA Statement Pointer before Executing a Program

It is good practice to restore the DATA statement pointer with an INIT statement or a RESTORE statement before a program is executed. For example, RUN 1ØØ does not restore the pointer, and if a program is run previous to the current program, then the DATA statement pointer might be pointing out to "right field" (or some obscure place). If the pointer does not point to a valid data item in the current program, then a read error occurs the first time a READ statement is executed in the program. Executing the RUN statement without specifying a line number does, however, restore the DATA statement pointer.

THE SAVE STATEMENT

Syntax Form:

[Line number] SAV [I/O address] [line number [, line number]]

Descriptive Form:

[Line number] SAVE [I/O address] [line number [starting , line number ending]]

Purpose

The SAVE statement transfers a copy of the current BASIC program to the specified output device. If an output device is not specified, then the internal magnetic tape unit is selected as the output device by default.

Explanation

Saving the Current Program on Magnetic Tape

If the SAVE statement is executed without specifying an I/O address or line numbers as parameters, then the BASIC interpreter transfers a complete copy of the current program to the internal magnetic tape unit when the statement SAVE is entered from the GS keyboard and the RETURN key is pressed.

The entire BASIC program is converted into an ASCII character string starting with the lowest line number and ending with the highest line number. Individual statements are separated with Carriage Return (CR) characters. The program is then sent to the internal magnetic tape unit as an ASCII character string. The SAVE operation does not disturb the assigned values of variables or the system environmental conditions. This means the SAVE statement can be executed as part of the program without disturbing the parameters of the program.

Before a SAVE statement is executed, the read/write head of the output device must be positioned at the beginning of a program file. For example, the following sequence transfers a copy of the current program to file number 7 on the internal magnetic tape unit.

```
FIND 7
SAVE
```

This example assumes, of course, file number 7 has been previously marked to a size large enough to store the program. (See the MARK statement in this section for details.)

Specifying an Output Device

A copy of the current program can be sent to any peripheral device in the system by specifying the appropriate primary address in the SAVE statement. For example:

```
200 SAVE @15:
```

This statement causes the BASIC interpreter to send a copy of the current program to device number 15 on the General Purpose Interface Bus. The entire program is converted to an ASCII character string with Carriage Returns separating each statement. The ASCII string is sent to device 15 one character at a time. Specifying the primary address is the only requirement for an I/O address. The BASIC interpreter automatically issues a secondary address 1 which tells the receiving device that the ASCII string represents a program to be saved.

Specifying a Line Number as a Parameter

If a line number is specified as a parameter, the BASIC interpreter sends the specified program line to the output device as an ASCII string. For example:

```
SAVE 200
```

This statement sends program line 200 to the internal magnetic tape unit as an ASCII string terminated by a Carriage Return (CR).

Specifying Two Line Numbers as Parameters

If two line numbers are specified as parameters, then the specified program lines and all program lines in between are sent to the output device. For example:

```
SAVE @10: 200,400
```

This statement causes the BASIC interpreter to send program lines 200 through 400 to peripheral device number 10 on the General Purpose Interface Bus.

Secret Programs Cannot be Saved

If a BASIC program is brought into memory from a "SECRET" program file, then the program cannot be saved or listed. The program can only be executed. If an attempt is made to SAVE a secret program after it is brought into memory, an error occurs and the appropriate error message is printed on the GS display.

A secret program can only be removed from memory by executing a DELETE ALL statement, an OLD statement, or by turning the power switch OFF. (See the SECRET statement in this section for details on making a program secret.)

THE SECRET STATEMENT

Syntax Form:

[Line number] SEC [I/O address]

Descriptive Form:

[Line number] SECRET [I/O address]

Purpose

The SECRET statement marks the current program secret. When a secret program is brought into memory from the internal magnetic tape unit or an external peripheral device, the program can only be executed; it can never be listed, saved, or in general output from memory.

Explanation

Creating a New Program and Marking it Secret

A new program can be marked secret any time during the course of program generation. For example, the secret status of a new program can be set before the program is entered into memory, while it is paritally entered into memory, or just before it is ready to be saved on an external media. A non-secret program can also be brought into memory, marked secret, and then output to a peripheral device with the SAVE statement.

The program currently in memory is marked secret by entering the keyword SECRET from the GS keyboard and pressing the RETURN key. A program marked in this way can be listed on the GS display and edited if necessary while it is still in memory; however, once a secret program is saved and then brought back into memory it can only be executed; it can never be listed or saved again.

Saving a New Secret Program

If a program is marked SECRET while it is still in memory, then the program can be saved on an external media like any other non-secret program. A new secret program is output from

memory with the SAVE statement. The program is output in a scrambled format and only the Graphic System has the ability to unscramble this format when the program is brought back into memory. If the program is stored on the internal magnetic tape, then the file header is marked SECRET when the program enters the file. The SECRET status of the file is automatically listed when a TLIST statement is executed.

Bringing a Secret Program Back into Memory

Secret programs are brought back into memory with the OLD statement, just like a non-secret program. If the program is brought in from the internal magnetic tape unit, then the file header tells the BASIC interpreter that the program is in the secret format and should be unscrambled. If the secret program is brought in from an external peripheral device, and the program does not have a file header, then SECRET must be set from the GS keyboard before the OLD operation is executed. This tells the BASIC interpreter that the program coming in is in the secret format and should be unscrambled.

NOTE

Any file APPENDed to a program which is SECRET will be treated as if it were SECRET.

Removing Secret Programs from Memory

There are three methods to remove a secret program from memory:

1. Execute a DELETE ALL statement directly from the GS keyboard.
2. Execute an OLD operation on a non-secret program.
3. Turn off the system power.

THE TLIST STATEMENT

Syntax Form:

[Line number] TLI [I/O address]

Descriptive Form:

[Line number] TLIST [I/O address]

Purpose

The TLIST statement causes the internal magnetic tape unit to list a tape file directory for the current magnetic tape cartridge. The list is sent to the GS display or to the specified external peripheral device. The list is displayed in directory format.

Explanation

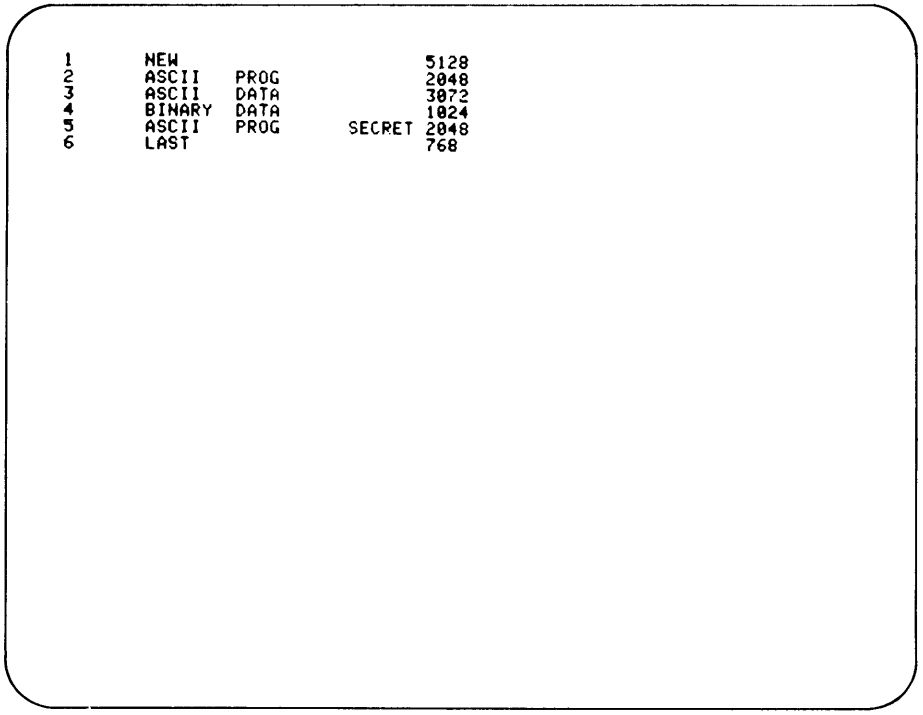
Listing the Tape Directory on the GS Display

When the TLIST statement is executed directly from the GS keyboard or under program control, the internal magnetic tape unit rewinds the magnetic tape. A fast search is then executed to find the file header for each file on the tape. As each file header is found, the header information is sent to the GS display as an ASCII character string terminated by a Carriage Return. The information is then printed on the GS display and the search for the next file header begins. The tape list operation is initiated by entering the keyword TLIST from the keyboard and pressing RETURN or by executing a TLIST statement under program control such as . . .

```
250 TLIST
```

A typical tape directory is shown below.

GS Display Output



Notice that for program files, the directory lists the tape file number, whether the program is secret or not secret, and the maximum number of bytes allocated to the file (physical length). For data files, the directory lists the file number, the type of file (ASCII or BINARY) and the maximum number of bytes allocated to the file. If the file is empty, it is listed as NEW, and if the file is the last (dummy) file, it is listed as the LAST file.

Sending the Tape Directory to an External Peripheral Device

If an I/O address is specified in the TLIST statement, then the tape directory is sent to the specified peripheral device as a series of ASCII character strings each terminated by a Carriage Return. For example:

```
260 TLIST @10:
```

When this statement is executed, the I/O address @10,19: is issued over the General Purpose Interface Bus (GPIB). Primary address 10 tells peripheral device number 10 that it has been selected to take part in the upcoming I/O operation. Secondary address 19 is issued by default and tells peripheral device 10 that the information coming next is a program or a tape directory to be listed. After the I/O address is issued, the tape directory is sent over the GPIB as a series of ASCII character strings. Each ASCII character string represents the information in one file header and is terminated by a Carriage Return character. It is up to device 10 to receive the information and print the information in directory format.

THE TYP FUNCTION

Syntax Form:

TYP numeric expression

Descriptive Form:

TYP logical unit number

Purpose

The TYP (Type) function returns an integer from Ø through 4 which indicates the kind of data stored as the next data item in the current internal magnetic tape file.

Explanation

The TYP function is primarily used to find out whether the next data item in a binary data file is numeric data, character string data, or an end of file mark. For example:

```
1ØØ T=TYP(Ø)
```

When this statement is executed, the BASIC interpreter looks at the next data item in the current magnetic tape file to see what kind of data resides there. The BASIC interpreter assigns a Ø to the variable T if the file is empty, a 1 to the variable T if the next item is an End Of File character, a 2 if the next item is numeric data or a character string formatted in ASCII code, a 3 if the next item is numeric data formatted in machine dependent binary code, and a 4 if the next item is a character string formatted in machine dependent binary code. (The Ø in parenthesis specifies logical unit Ø which is the internal magnetic tape unit.) The following table summarizes the meaning of each integer returned by the TYP function:

Ø	Empty File or File Not Open
1	End of File Character
2	Numeric Data or Character String Data/ASCII Format
3	Numeric Data/Binary Format
4	Character String Data/Binary Format

It is important when reading binary data files with the READ statement that the variable specified as the target to receive the data item is of the correct type; that is, a numeric variable must be specified for numeric data and a string variable must be specified for character strings. If a numeric variable is specified in a READ statement and the next data item to be read is a character string, then an error occurs and program execution is aborted. If the next data item type is unknown, then the TYP function offers the only means to find out what the data type is without terminating program execution. The following is a general purpose program which uses the TYP function to list the contents of any magnetic tape file on the GS display.

```
100 INIT
110 PRINT "LEnter a File Number and Press RETURN";
120 INPUT F
130 FIND F
140 T=TYP(Ø)
150 GO TO T OF 270,180,210,240
160 PRINT "EMPTY FILE"
170 END
180 INPUT @33:A$
190 PRINT A$
200 GO TO 140
210 READ @33:A
220 PRINT A
230 GO TO 140
240 READ @33:A$
250 PRINT A$
260 GO TO 140
270 PRINT "END OF FILE"
280 FIND F
290 END
```

Line 100 in this program initializes the system environmental parameters. Line 110 then prints the message "Enter a File Number and Press RETURN." After a file number is entered and the RETURN key pressed, the INPUT statement in line 120 assigns the entry to the variable F. Line 130 positions the internal magnetic tape read/ write head to the beginning of the specified file and the TYP function in line 140 examines the first data item in the file.

Line 150 then transfers program control to one line number in the line number list. The decision to transfer control is based on the value of T.

If the tape file is a new (empty) file, the TYP function returns a 0 to the variable T. Line 150 then transfers program control to line 160 where the message "EMPTY FILE" is printed on the GS display and program control is terminated in line 170.

If the file contains information stored in ASCII format (either program lines or data), the TYP function in line 140 returns a 2 to the variable T. This causes line 150 to transfer program control to line 180, the second line number in the line number list.

Line 180 assigns the first data item in the file to A$. Line 190 then prints the data item on the GS display. Program control is then transferred back to line 140 when line 200 is executed, and the process is repeated on the second data item in the file.

If the specified file is a binary file and the first data item is a numeric value, the TYP function in line 140 returns a 3 to the variable T. This causes line 150 to transfer program control to line 210, the third line number in the list. Line 210 assigns the value to the numeric variable A. The value is printed on the GS display in line 220. In line 230, program control is transferred to line 140 and the process is repeated on the next data item in the file.

If the second data item in the binary file is a character string, the TYP function in line 140 returns a 4 to the variable T. This causes line 150 to transfer program execution to line 240, the fourth line number in the line number list. Line 240 assigns the character string to the variable A$ and the character string is printed on the GS display when line 240 is executed. Line 260 returns program execution to line 140 and the process is repeated on the next data item in the file.

When the End Of File character is reached in the file, the TYP function in line 140 returns a 1 to the variable T. This causes line 150 to transfer program execution to line 270, the first line number in the line number list. Line 270 prints the message "END OF FILE" on the GS display. The magnetic tape head is positioned to the beginning of the file in line 280 and the program is terminated in line 290.

This program illustrates the use of the TYP function and can be used to list the contents of any file on the internal magnetic tape (except a SECRET program file).

THE WBYTE STATEMENT

Syntax Form:

[Line number] WBY [@ numeric expression [, numeric expression] ... :]

[numeric expression] [,numeric expression] ...

Descriptive Form:

[Line number] WBYTE [@ absolute address [, absolute address] ... :]

[data bytes to be sent out over the General Purpose Interface Bus]

Purpose

The WBYTE (Write Byte) statement is used to send 8-bit data bytes to an external peripheral device on the General Purpose Interface Bus. This statement gives the GS keyboard operator complete control over the eight lines on the GPIB Data Bus and control over the ATN (Attention) signal line and the EOI (End or Identify) signal line on the GPIB Management Bus.

Explanation

The WBYTE statement gives the GS keyboard operator and the BASIC program direct access to the eight data lines on the GPIB. Normally, this statement is used to address and control peripheral devices which require special bit patterns for interface circuitry control.

One or more bytes can be sent to an external peripheral device over the GPIB using the WBYTE statement. If the ATN signal line is activated as a byte is transferred, the byte is treated as a peripheral address or controller command. Data bytes sent after ATN goes inactive represent numeric data, character string data, or special peripheral instructions.

Reviewing the GPIB Hardware Features

If you're unfamiliar with the General Purpose Interface Bus operation, at this point it is appropriate to review the hardware features of the bus. A discussion of the hardware features is found in Appendix C.

An Overview

In general, each data byte to be sent over the General Purpose Interface Bus is specified as an integer from −255 through +255 in the WBYTE statement. Integers Ø through 255 represent the decimal equivalent of the data byte to be transferred. Negative values specify that the EOI signal line is to be activated as the data byte is transferred. Specifying −Ø is not allowed. The decimal equivalent of a data byte can be specified as a numeric expression as long as the BASIC interpreter can reduce the numeric expression to a numeric contant and round the constant to an integer within the range −255 through +255.

Data bytes specified between the "at" sign (@) and the colon (:) are issued when the ATN (Attention) signal line is activated. These data bytes are treated as peripheral addresses and controller commands. Data bytes which follow the colon in the WBYTE statement are issued with ATN inactive and are treated as data.

Lower Level I/O addresses and Control Commands

Up to now, primary addresses have been specified in I/O statements as peripheral device numbers. For example:

```
INPUT @15:A,B,C$
```

This statement selects peripheral device number 15 on the GPIB as the input source. Internally, the BASIC interpreter converts primary address 15 to one of two lower level primary addresses; a primary talk address or a primary listen address. In this case, the primary talk address for device 15 is issued because an INPUT statement is being executed. If the keyword were PRINT instead of INPUT, then the BASIC interpreter would issue the primary listen address instead of the primary talk address. In each case, the BASIC interpreter issues the lower level primary address appropriate for the keyword. The lower level address is called the "absolute" primary address and can be issued directly to a peripheral device via the WBYTE statement.

The following table lists the absolute primary talk and listen addresses for each peripheral device on the GPIB. These absolute addresses are specified as a decimal number in the WBYTE statement and must be issued with the ATN signal line activated; otherwise, the addresses are treated as data. The binary bit pattern for each address is also given in the table.

The table which follows the Primary Address table lists each secondary addresses for the system with its decimal equivalent and binary bit pattern.

GPIB PRIMARY ADDRESSES																	
PERIPHERAL DEVICE NUMBER	PRIMARY LISTEN ADDRESS									PRIMARY TALK ADDRESS								
	DECIMAL VALUE	DIO BUS								DECIMAL VALUE	DIO BUS							
		8	7	6	5	4	3	2	1		8	7	6	5	4	3	2	1
Device 0	32	0	0	1	0	0	0	0	0	64	0	1	0	0	0	0	0	0
Device 1	33	0	0	1	0	0	0	0	1	65	0	1	0	0	0	0	0	1
Device 2	34	0	0	1	0	0	0	1	0	66	0	1	0	0	0	0	1	0
Device 3	35	0	0	1	0	0	0	1	1	67	0	1	0	0	0	0	1	1
Device 4	36	0	0	1	0	0	1	0	0	68	0	1	0	0	0	1	0	0
Device 5	37	0	0	1	0	0	1	0	1	69	0	1	0	0	0	1	0	1
Device 6	38	0	0	1	0	0	1	1	0	7Ø	0	1	0	0	0	1	1	0
Device 7	39	0	0	1	0	0	1	1	1	71	0	1	0	0	0	1	1	1
Device 8	4Ø	0	0	1	0	1	0	0	0	72	0	1	0	0	1	0	0	0
Device 9	41	0	0	1	0	1	0	0	1	73	0	1	0	0	1	0	0	1
Device 10	42	0	0	1	0	1	0	1	0	74	0	1	0	0	1	0	1	0
Device 11	43	0	0	1	0	1	0	1	1	75	0	1	0	0	1	0	1	1
Device 12	44	0	0	1	0	1	1	0	0	76	0	1	0	0	1	1	0	0
Device 13	45	0	0	1	0	1	1	0	1	77	0	1	0	0	1	1	0	1
Device 14	46	0	0	1	0	1	1	1	0	78	0	1	0	0	1	1	1	0
Device 15	47	0	0	1	0	1	1	1	1	79	0	1	0	0	1	1	1	1
Device 16	48	0	0	1	1	0	0	0	0	8Ø	0	1	0	1	0	0	0	0
Device 17	49	0	0	1	1	0	0	0	1	81	0	1	0	1	0	0	0	1
Device 18	5Ø	0	0	1	1	0	0	1	0	82	0	1	0	1	0	0	1	0
Device 19	51	0	0	1	1	0	0	1	1	83	0	1	0	1	0	0	1	1
Device 20	52	0	0	1	1	0	1	0	0	84	0	1	0	1	0	1	0	0
Device 21	53	0	0	1	1	0	1	0	1	85	0	1	0	1	0	1	0	1
Device 22	54	0	0	1	1	0	1	1	0	86	0	1	0	1	0	1	1	0
Device 23	55	0	0	1	1	0	1	1	1	87	0	1	0	1	0	1	1	1
Device 24	56	0	0	1	1	1	0	0	0	88	0	1	0	1	1	0	0	0
Device 25	57	0	0	1	1	1	0	0	1	89	0	1	0	1	1	0	0	1
Device 26	58	0	0	1	1	1	0	1	0	9Ø	0	1	0	1	1	0	1	0
Device 27	59	0	0	1	1	1	0	1	1	91	0	1	0	1	1	0	1	1
Device 28	6Ø	0	0	1	1	1	1	0	0	92	0	1	0	1	1	1	0	0
Device 29	61	0	0	1	1	1	1	0	1	93	0	1	0	1	1	1	0	1
Device 30	62	0	0	1	1	1	1	1	0	94	0	1	0	1	1	1	1	0
UNLISTEN/UNTALK	63	0	0	1	1	1	1	1	1	95	0	1	0	1	1	1	1	1

GPIB SECONDARY ADDRESSES									
SECONDARY ADDRESS	PREDEFINED MEANING	DECIMAL VALUE	DATA BUS						
			8	7	6	5	4	3	2	1
Ø	"STATUS"	96	0	1	1	0	0	0	0	0
1	SAVE	97	0	1	1	0	0	0	0	1
2	CLOSE	98	0	1	1	0	0	0	1	0
3	OPEN	99	0	1	1	0	0	0	1	1
4	OLD/APPEND	1ØØ	0	1	1	0	0	1	0	0
5	CREATE	1Ø1	0	1	1	0	0	1	0	1
6	TYPE	1Ø2	0	1	1	0	0	1	1	0
7	KILL	1Ø3	0	1	1	0	0	1	1	1
8	UNIT	1Ø4	0	1	1	0	1	0	0	0
9	DIRECTORY	1Ø5	0	1	1	0	1	0	0	1
1Ø	COPY	1Ø6	0	1	1	0	1	0	1	0
11	RELABEL	1Ø7	0	1	1	0	1	0	1	1
12	PRINT	1Ø8	0	1	1	0	1	1	0	0
13	INPUT	1Ø9	0	1	1	0	1	1	0	1
14	READ	11Ø	0	1	1	0	1	1	1	0
15	WRITE	111	0	1	1	0	1	1	1	1
16	ASSIGN	112	0	1	1	1	0	0	0	0
17	"ALPHASCALE"	113	0	1	1	1	0	0	0	1
18	FONT	114	0	1	1	1	0	0	1	0
19	LIST/TLIST	115	0	1	1	1	0	0	1	1
2Ø	DRAW/RDRAW	116	0	1	1	1	0	1	0	0
21	MOVE/RMOVE	117	0	1	1	1	0	1	0	1
22	PAGE	118	0	1	1	1	0	1	1	0
23	HOME	119	0	1	1	1	0	1	1	1
24	GIN	12Ø	0	1	1	1	1	0	0	0
25	"ALPHAROTATE"	121	0	1	1	1	1	0	0	1
26	STATUS	122	0	1	1	1	1	0	1	0
27	FIND	123	0	1	1	1	1	0	1	1
28	MARK	124	0	1	1	1	1	1	0	0
29	SECRET	125	0	1	1	1	1	1	0	1
3Ø	"ERROR"	126	0	1	1	1	1	1	1	0
31	DASH	127	0	1	1	1	1	1	1	1

How the RBYTE and WBYTE Statements Work Together

The RBYTE (Read Byte) and WBYTE (Write Byte) statements provide direct access to the General Purpose Interface Bus from the GS keyboard. Normally, the WBYTE statement is used to assign an external peripheral device as a listener or a talker. If the peripheral device is assigned a listener, then the data to be sent is normally specified in the same WBYTE statement. Another WBYTE statement is used to terminate the transfer by sending the universal command UNLISTEN to the peripheral device. If the peripheral device is assigned the role of a talker and the BASIC interpreter intends to listen, then the RBYTE statement usually follows next in the BASIC program to assign the incoming data bytes to numeric variables (one byte per variable).

Data Bytes on the Data Bus

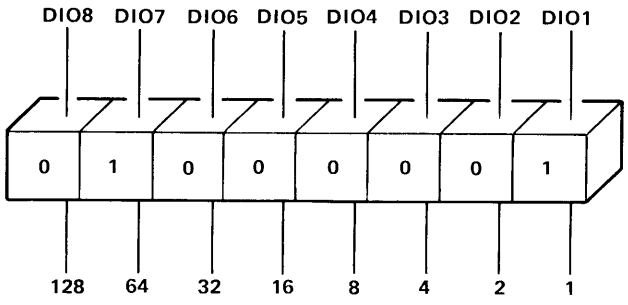
The eight-line Data Bus (within the GPIB) is used to transfer eight bits of binary information at a time. This includes transfers from one peripheral device to another peripheral device, as well as transfers between a peripheral device and Graphic System random access memory. Each 8-bit binary pattern is called a byte.

Binary-to-Decimal Conversion

The following information is provided for those of you who are unfamiliar with the binary-to-decimal conversion process. For those of you who are familiar with the technique, go on to the next topic.

There can only be 256 unique 8-bit patterns transferred over the GPIB because the Data Bus is only 8 lines wide. In the WBYTE statement, each 8-bit pattern (byte) is represented by a decimal number within the range Ø through 255. If the decimal number is preceded by a minus sign, then the EOI (End or Identify) signal line on the GPIB Management Bus is activated when the byte is transferred.

Data bytes received by the BASIC interpreter over the GPIB are converted to their decimal equivalents before they are assigned to numeric variables. The following illustration shows how an eight-bit binary code is converted to a decimal number.



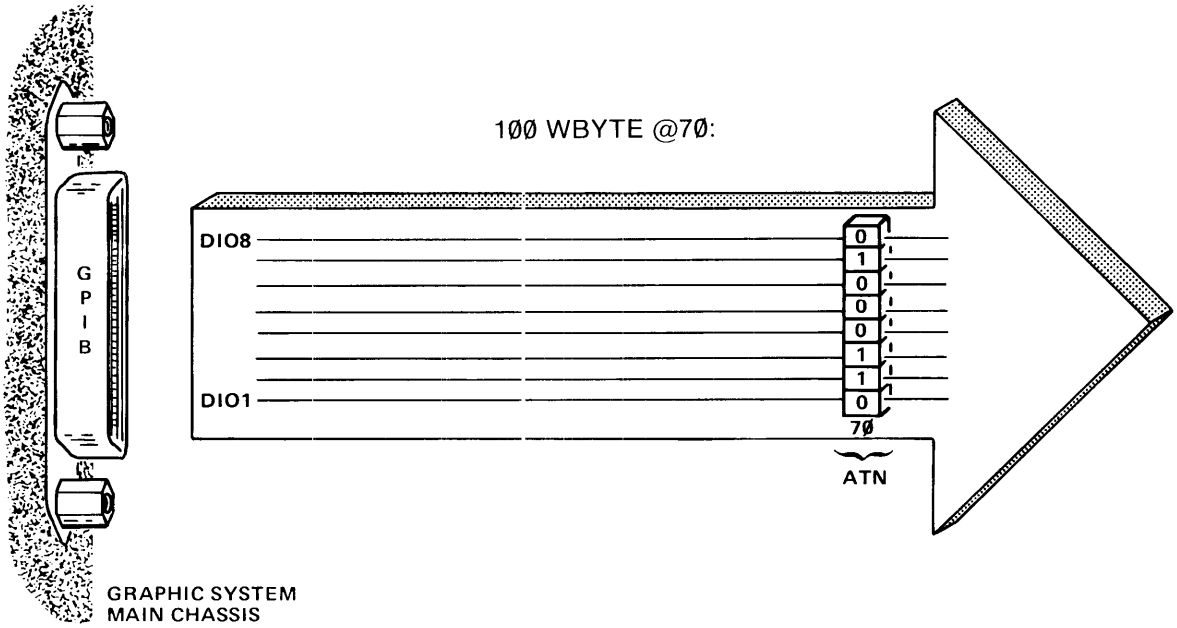
64 + 1 = 65

Each line on the Data Bus represents a binary bit. If the line is electrically high (near +5 Vdc), the line represents a binary Ø. If the line is electrically low (near Ø Vdc), the line represents a binary 1. In the illustration above, DIO1 and DIO7 are electrically low, so they represent a binary 1. The rest of the lines are electrically high; they represent a binary Ø.

Each bit in the eight-bit code carries a decimal weight as shown under each block in the illustration. The right-most bit (DIO1) carries a weight of 1; the seventh bit (DIO7) carries a weight of 64. Adding these weights together gives the decimal equivalent of the byte. In this case, the byte is equivalent to 65 (base1Ø).

Issuing Primary Addresses

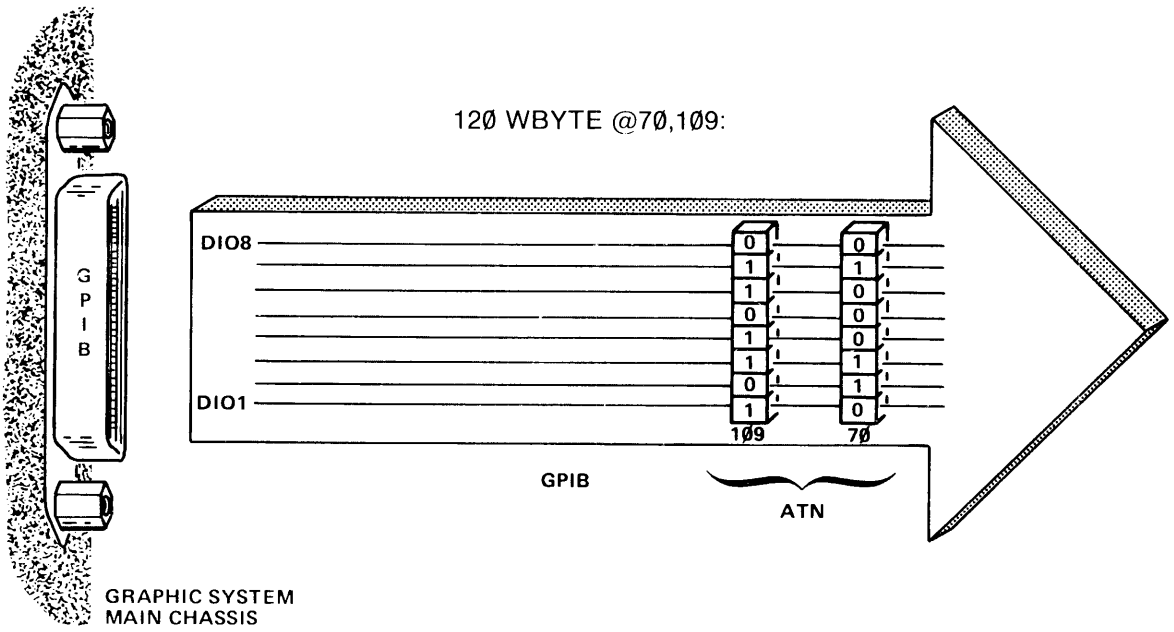
The following example illustrates how a peripheral device on the GPIB is addressed using the WBYTE statement:



When line 1ØØ is executed under program control, the BASIC interpreter sends the absolute primary address 7Ø out the back door. The "at" sign (@) in the statement causes the BASIC interpreter to activate the ATN signal line on the GPIB Management Bus. This tells each peripheral device on the GPIB to "sit up and listen because the data bytes to follow are peripheral addresses and control commands." The primary talk address for device 6 (decimal 7Ø) is then sent over the GPIB Data Bus. This tells device 6 that it has been selected to take part as a talker in the upcoming data transfer. The colon in the WBYTE statement causes the BASIC interpreter to release the ATN signal line. Because device 6 is the only peripheral device addressed while ATN is active low, device 6 is the only device that can take part in the transfer. All other peripheral devices on the GPIB are prevented from listening to the bus or talking to the bus.

Issuing Secondary Addresses

Normally, a secondary address is issued after the primary address to tell the peripheral device what the data transfer is all about. The secondary address, if issued, must immediately follow the primary address. Both the primary address and the secondary address must be specified between the "at" sign (@) and the colon (:). This ensures that the ATN signal line is activated during the transfer. Otherwise the data bytes are interpreted as data instead of peripheral addresses. For example:

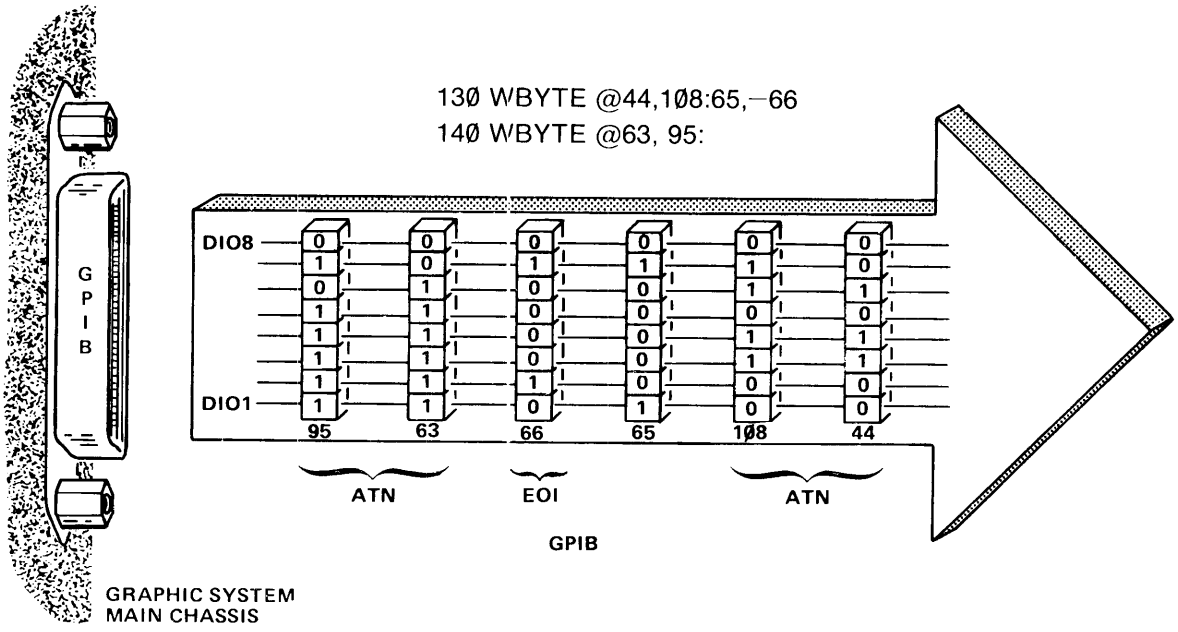


When statement 120 is executed, the primary talk address 70 and the secondary address 109 are sequentially issued over the GPIB with the ATN signal line activated. Primary talk address 70 tells peripheral device number 6 that it has been selected as the talker for the upcoming data transfer. Secondary address 109 is also issued over the GPIB while the ATN signal line is activated. Secondary address 109 tells device number 6 that the BASIC interpreter is executing function number 13 or an INPUT statement and to start sending ASCII data strings. Device number 6 assumes this secondary address has this meaning if the peripheral is designed to conform to the predefined meanings for the Graphic System secondary addresses. Actually, the peripheral device can be designed to interpret secondary addresses in any way the designer chooses. (A detailed explanation of the predefined meaning of each secondary address is given in Appendix B.)

Normally, a statement like this is followed by a RBYTE (Read Byte) statement which causes the BASIC interpreter to receive the data bytes from peripheral device 6. One or more peripheral devices on the GPIB can also be assigned as listeners in the WBYTE statement to receive the data bytes from peripheral number 6. (For detailed information on receiving data bytes via the RBYTE statement over the GPIB, refer to the RBYTE statement in this section.)

Transferring Data Bytes with the WBYTE Statement

If data bytes are to be transferred to a peripheral device after the device is addressed, then the data bytes are specified after the colon (:) in the WBYTE statement. For example:



When statement 13Ø is executed, the ATN line on the GPIB is activated and the primary listen address for device number 12 (decimal 44) is sent over the GPIB followed by the secondary address 12 (decimal 1Ø8). The ATN signal line is then released. The primary listen address (decimal 44) tells device number 12 that it has been selected as a listener for the upcoming data transfer. Secondary address 12 (decimal 1Ø8) tells device number 12 to prepare to receive ASCII data.

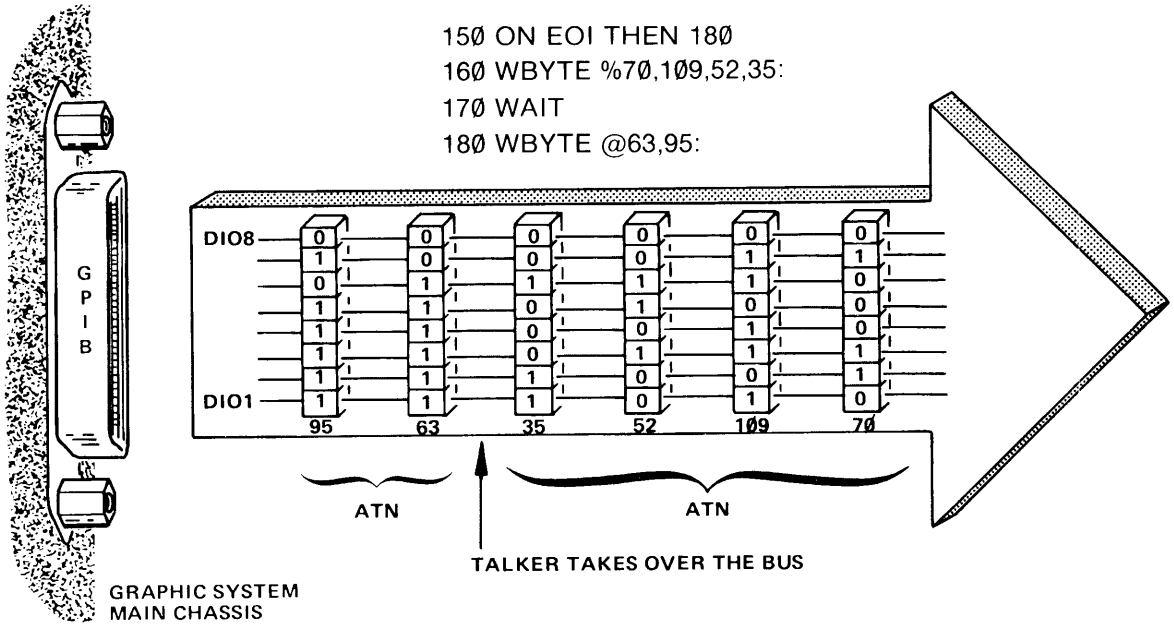
When the ATN signal line is released, only device 12 can listen to the GPIB because it is the only device addressed to take part in the transfer. The data byte decimal 65 is sent over the GPIB and received by device number 12. The data byte decimal 66 follows next. Because the ATN signal line is inactive while these data bytes are transferred, the bytes are treated as data and not peripheral addresses. Notice that the second data byte (decimal 66) is specified as a negative value in the WBYTE statement in line 13Ø. This causes the BASIC interpreter to activate the EOI (End or Identify) signal line on the GPIB which tells the listener that decimal 66 is the last byte to be transferred.

After decimal 66 is transferred, the BASIC program continues to line 14Ø. In line 14Ø, the ATN signal line is again activated and the universal commands UNTALK (decimal 95) and UNLISTEN (decimal 63) are issued over the bus. This places every active peripheral device on the GPIB in a known quiescent state and terminates the transfer.

It is not always necessary to specify the last data byte as negative to activate EOI and it's not always necessary to issue an UNTALK and UNLISTEN command at the end of a transfer; however, it is a good bus management practice to do so. The UNTALK and UNLISTEN commands are issued automatically by the BASIC interpreter after the keywords PRINT, INPUT, GIN, DRAW, etc. are executed.

Peripheral to Peripheral Transfers

Sometimes it's desirable to have one peripheral device talk to another peripheral device on the GPIB without involving the Graphic System controller. For example, you might want to transfer data from a paper tape reader to a line printer or a storage device. The following example illustrates how one peripheral device is assigned to a talker role and several other peripheral devices are assigned a listener role. This sequence starts a direct transfer from one peripheral device to another peripheral device.



When line 15Ø is executed under program control, the EOI (End or Identify) interrupt facility is activated. This tells the BASIC interpreter to be on the lookout for an active EOI signal line on the GPIB. When EOI goes active at a later time, the BASIC interpreter transfers program control to line 18Ø.

Line 16Ø is executed next. Primary talk address 7Ø tells peripheral device number 6 that it is going to be the talker for the upcoming data transfer. Secondary address 1Ø9 tells device 6 to start sending ASCII data over the GPIB as soon as the ATN signal line is released. The primary listen address for device 2Ø is issued next (decimal 52), followed by primary listen address for device 3 (decimal 35). This tells peripheral device 2Ø and peripheral device 3 to start listening to the talker when the ATN signal line is released. The percent sign (%) is specified in this case instead of the "at" sign (@); this tells the BASIC interpreter to get off the bus and let the assigned talker take over when the ATN line is released. The colon in the WBYTE statement (:) causes the BASIC interpreter to release ATN.

At this point, the talker (device number 6) takes over the bus and starts sending data to device number 2Ø and device number 3. The BASIC program in the meantime goes to line number 17Ø and waits patiently for the talker to activate the EOI signal line as the last byte of data is transferred over the bus. When EOI is activated (for at least 350 μs), program control is transferred to line 18Ø where the BASIC interpreter activates the ATN signal line and issues the universal commands UNTALK/UNLISTEN over the GPIB. This places all active peripheral devices on the GPIB in a known quiescent state and terminates the I/O operation.

When specifying primary and secondary addresses in a WBYTE statement, only one peripheral device can be assigned as a talker, but up to 14 peripheral devices can be assigned as listeners. Each primary address can be followed by one or more secondary addresses. The primary addresses can be specified in any order; it doesn't matter. Universal controller commands can also be thrown in.

Refer to the RBYTE statement in this section for information on how to receive data bytes from a peripheral device over the GPIB.

THE WRITE STATEMENT

Syntax Form:

[Line number] WRI [I/O address] { string constant / string variable / numeric expression }

[[, { string constant / string variable / numeric expression }] ...]

Descriptive Form:

[Line number] WRITE [I/O address] data item to be written in machine dependent binary code [, data item to be written in machine dependent binary code] ...

PURPOSE

The WRITE statement sends the specified data items to the specified peripheral device in machine dependent binary code. If a peripheral device is not specified, the internal magnetic tape unit is selected as the peripheral device by default.

EXPLANATION

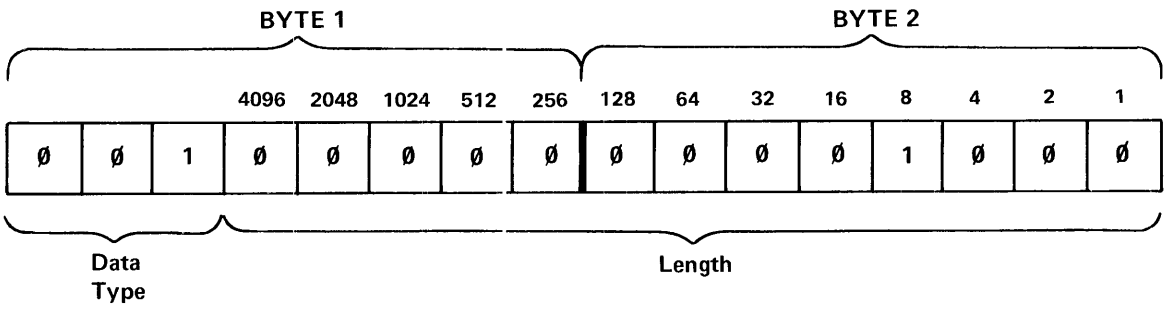
Machine Dependent Binary Code Defined

The term "machine dependent binary code" refers to the internal binary format used by the Graphic System to store BASIC programs and data in the random access memory . Data transferred to and from a peripheral device using the WRITE and READ statements is transferred in this internal format. Normally, transfers of this nature are faster, because the conversion back and forth between binary format and ASCII format is eliminated.

Transfers between the random access memory and a peripheral device in machine dependent binary code implies that the peripheral device receiving the information is able to understand the internal format used by the Graphic System. Normally, transfers of this nature are carried on with an external mass storage device which only stores the code.

Binary Format

Each data item transferred in machine dependent binary code is preceded by a two byte (16 bit) header. This applies whether the data item is numeric data or character string data. The two byte header contains information which tells the receiving device what the data item type is (numeric, string, or End of File mark) and the data item length (in bytes). The following diagram illustrates how information is stored in the two byte header:



Data Type

CODE	MEANING
Ø Ø Ø	Undefined
Ø Ø 1	Numeric Value
Ø 1 Ø	Character String
Ø 1 1	Undefined
1 Ø Ø	Undefined
1 Ø 1	Undefined
1 1 Ø	Undefined
1 1 1	End of File

The three high order bits of the first byte represent a binary number from Ø through 7. This number indicates the data item type. A binary 1 (ØØ1) means the data item is a numeric value. Binary 2 (Ø1Ø) means the data item is a character string. And binary 7 (111) means the data item is an End of File character. All other bit combinations are undefined.

The five remaining low order bits in the first byte of the header and the eight bits of the second byte of the header form a binary number which tells how many bytes are in the data item. Each bit carries a decimal weight as shown in the illustration. In this case, the data type is numeric (ØØ1) and the body of the data item (which follows the header) is eight bytes long.

As a general rule, each numeric data item is eight bytes long with a two byte header for a total length of 1Ø bytes. Each character string requires one byte for each character plus the two byte header and a system byte for a total length of LEN+3 (where LEN is the Length function).

Sending Binary Data to an Internal Magnetic Tape File

Binary data can be stored in a NEW file (one just created with the MARK statement), an ASCII file if it is first killed with the KILL statement, or an old binary file. Before data is sent to a file with the WRITE statement, the tape head on the internal tape unit must be positioned to the beginning of the file. For example:

```
1ØØ FIND 2
11Ø WRITE X,Y$,"PLOT 5Ø",1975
```

When line 1ØØ is executed, the tape head is positioned to the beginning of the storage area in file 2 (just past the file header). Line 11Ø then sends four data items to the file in machine dependent binary code. The numeric value assigned to the variable X is sent first, followed by the character string assigned to Y$, followed by the character string "PLOT 5Ø", followed by the numeric value 1975.

This statement illustrates how both numeric data and character string data can be sent to the internal tape with the same WRITE statement. The data items can be specified in any order.

Like the PRINT statement, once the data items are stored on tape, they become completely unattached to the variable symbols they are assigned to when they were stored in memory. This means that the data items can be assigned to any variable symbol when they are brought back into memory from the tape.

Writing Matrices

If an array variable is specified in a WRITE statement, then the elements of the array are sent to the specified peripheral device in machine dependent binary code. The elements of the array are issued one after another in row major order; each element takes 1Ø bytes of storage.

Sending Binary Data to a Partially-Filled Data File

Binary data can be stored in a partially-filled binary file, but first the tape head should be positioned to the EOF (End of File) character before the data is sent to the tape. This preserves the data already in the file. If the read/write head is positioned to the beginning of the file when the additional information is sent to the tape, the data already stored in the file is overwritten by

the new data coming in. The following program illustrates how to position the read/write head to the End Of File character in a partially-filled binary file. Additional data items are then added to the file.

The program is divided into two parts. The first part (lines 100 - 240) positions the tape head to the EOF mark in the middle of a binary data file. The second part (lines 250 - 340) adds additional data items to the file.

```
100 INIT
110 PAGE
120 PRINT "Enter a Binary File Number and Press RETURN":
130 INPUT F
140 FIND F
150 T=TYP(0)
160 GO TO T OF 250,230,210,190
170 FIND F
180 GO TO 250
190 READ @33:A$
200 GO TO 150
210 READ @33:A
220 GO TO 150
230 PRINT "ASCII Data File - Try Again"
240 GO TO 120
250 PRINT "What's for Supper ? ";
260 INPUT S$
270 WRITE S$
280 PRINT "Any Thing Else (yes or no)"
290 INPUT C$
300 IF C$="no" THEN 340
310 PRINT "What"
320 INPUT S$
330 GO TO 270
340 END
```

The first part of the program starts with line 100. Line 100 initializes the system environmental parameters, line 110 clears the GS display, and line 120 asks the keyboard operator which file he wants to access. When an entry is made and the RETURN key pressed, line 130 assigns the entry to the variable F. Line 140 then locates the specified file. At this point in the program, the TYP function is used to determine what kind of file is found.

If the file is new, then the TYP function in line 15Ø returns a Ø to the variable T. In line 16Ø, the value of T is used to indicate which line number in the line number list is executed next. Since T is equal to Ø, program execution continues to line 17Ø which repositions the tape head to the beginning of the file. Line 18Ø then passes control to line 25Ø where the program prepares to input data from the keyboard and write the data to the file in binary format.

If the specified file is an ASCII file, the TYP function returns a 2 to the variable T in line 15Ø. This causes line 16Ø to pass control to line 23Ø, the second line number in the line number list. Line 23Ø lets the keyboard operator know that the file is an ASCII file, and line 24Ø returns program control to 12Ø. Line 12Ø asks for another file number.

If the specified file is a binary file, the TYP function in line 15Ø returns either a 3 or a 4 depending on whether the first data item is a numberic value or a character string. If the first data item is a numeric value, the TYP function returns a 3 to the variable T. This causes line 16Ø to pass control to line 21Ø and the data item is read. The purpose for this statement is to advance the read head to the next data item in the binary file. Line 22Ø then passes control back to line 15Ø and the next data item in the file is examined. If the next data item is a character string, the TYP function returns a 4 to the variable T. This causes line 16Ø to transfer control to line 19Ø, the fourth line number specified in the line number list. Line 19Ø assigns the character string to A$. This action advances the tape head to the third data item in the file. Control is then passed back to line 15Ø, and the next data item in the file is examined. This process continues until the End of File character is found.

When the tape head is positioned over the End of File mark in the binary file, the TYP function in line 15Ø returns a 1 to the variable T. This causes line 16Ø to transfer control to line 25Ø and the program is ready to store additional data items.

The second part of the program adds data items to the partially-filled binary file. With the tape head positioned at the end of the last data item in the file, line 25Ø asks the question "What's for Supper?" The keyboard operator enters a menu item and presses the RETURN key. The entry is assigned to S$ in line 26Ø. The item is then sent to the tape in line 27Ø as a binary data item. The program then asks if there is anything else for supper (line 28Ø). If the answer is no, line 3ØØ transfers control to line 34Ø, an END statement. It is important to execute an END statement here to close the file. This makes sure that any data remaining in the magnetic tape memory buffer is "dumped" onto the tape. (A FIND statement or CLOSE statement can also be executed to close a file.)

If the answer to the question in line 28Ø is something besides "no," then program execution continues to line 31Ø where the questior "What" is asked. The entry is recorded and assigned to S$ in line 32Ø. Line 33Ø then transfers control to line 27Ø where the entry is sent to magnetic tape as a binary data item. This process continues until the keyboard operator enters "no" in response to the question in line 28Ø. The file is then closed and program execution is terminated.

Sending Binary Data to an External Peripheral Device

If an I/O address is specified in a WRITE statement, the specified data items are sent to the specified peripheral device in machine dependent binary code. For example:

```
25Ø  WRITE @14:56.9,"Data Base", 135.ØØ9
```

When this statement is executed, the I/O address @14,15: is sent over the General Purpose Interface Bus (GPIB). Primary address 14 tells peripheral device 14 that it has been selected to take part in an I/O operation. Secondary address 15 is issued by default and tells device 14 that the BASIC interpreter is executing a WRITE statement. Device 14 then prepares to receive data items in machine dependent binary code. After the I/O address is sent over the GPIB, the BASIC interpreter sends the specified data items to device 14 in machine dependent binary code. It is up to device 14 to receive the data items and either store the items or process the items.

Computing External Storage Requirements for Binary Data

The following guidelines can be used to compute the external storage requirements for binary data:

1. Each numeric value requires 13 bytes of storage space.

2. Each Character string requires LEN+18 bytes of storage space, where LEN represents the number of characters in the string.

3. A numeric array requires 8 bytes of storage for each element, plus 18 bytes for the label.

MATH OPERATIONS

Section 8

MATH OPERATIONS

INTRODUCTION TO MATH OPERATIONS

There are nine standard math functions, six trigonometric functions, six matrix functions, and twenty-six user-definable functions ava lable for use in math operations.

Standard Math Functions

FUNCTION	PURPOSE
ABS	Returns the absolute value of the specified numeric expression.
EXP	Returns the value of e (the natural logarithm base) raised to the power specified by the numeric expression.
INT	Returns the largest integer possible without exceeding the specified numeric expression.
LGT	Returns the logarithm of the specified numeric expression to the base 1Ø.
LOG	Returns the logarithm of the specified numeric expression to the base e (the natural logarithm base).
PI	Returns the value 3.14159265359.
RND	Returns a random number between Ø and 1.
SGN	Returns a +1 if the specified numeric expression is positive, a Ø if the specified numeric expression is zero, and a −1 if the specified numeric expression is negative.
SQR	Returns the square root of the specified numeric expression.

Trigonometric Functions

Trigonometry provides a method of finding the values of the remaining angles and sides of a right triangle when only a few angles and sides are known. There are six trigonometric functions available to solve trigonometric problems. These functions are SIN (Sine), COS (Cosine), TAN (Tangent), ASN (Arc Sine), ACS (Arc Cosine), and ATN (Arc Tangent).

Matrix Functions

The DET function returns the determinant of the square part of the last matrix evaluated by the INV function.

The IDN routine returns a matrix whose square part is an identity matrix.

The INV function returns the inverse and solutions to systems of linear equations of a matrix.

The MPY function returns the matrix multiplication of two matrices.

The SUM Function algebraically adds the elements of the specified array and returns the algebraic sum.

The TRN function returns the transpose of a matrix.

User-Definable Functions

In addition to the standard math functions, trigonometric functions, and matrix function, the DEF FN statement allows up to twenty-six numeric expressions to be defined as numeric functions. Defining a numeric expression in this fashion is convenient when a numeric expression must be specified repeatedly throughout a BASIC program.

THE ABS FUNCTION

Syntax Form:

ABS numeric expression

Purpose

The ABS (Absolute Value) function returns the absolute value of the specified numeric expression.

Explanation

Immediate Execution

The BASIC interpreter returns the absolute value of a numeric expression if the ABS function is entered directly from the GS keyboard and the RETURN key is pressed. The absolute value is returned to the GS display one line below the entry statement. For example:

```
ABS (4*9/6-8)
  2
```

When this expression is evaluated, the parameter 4*9/6-8 is reduced to the numeric constant −2. The absolute value of −2 is returned to the GS display one line below the entry statement.

Program Execution

The ABS function returns a numeric result and can be specified as part of a numeric expression. The following examples illustrate how the ABS function can be specified as part of a numeric expression and evaluated under program control:

```
100 LET X=ABS (X+Y)
110 PRINT ABS (45−78+13)
120 WRITE 21X/ABS(3* Y)+5
```

THE ACS FUNCTION

Syntax Form:

{ACOS / ACS} numeric expression

Purpose

The ACS (Arc Cosine) function reduces the specified numeric expression to a numeric constant, interprets the numeric constant as the cosine of an angle, and returns the angle expressed in the current trigonometric units for the system.

Explanation

ACS is the Opposite of COS

The ACS function is the opposite of the COS function. The COS function requires an angle for a parameter and returns the cosine of the angle. The ACS function, on the other hand, requires the cosine of an angle for a parameter, and returns the corresponding angle expressed in the current trigonometric units for the system. For example, the cosine of 34 degrees is .829; the arc cosine of .829 is 34 degrees. The ACS parameter must be specified within the range −1 through +1 or an error occurs and the appropriate error message is printed on the GS display.

Immediate Execution

The BASIC interpreter returns the angle of any value between −1 and 1 immediately, if the ACS function and the value is entered from the GS keyboard and the RETURN key is pressed. The BASIC interpreter assumes the value is the cosine of an angle measured in radians unless the system environment is set to degrees or grads. (Refer to the SET statement in the Environmental Control section for details.) The angle is returned to the GS display and printed one line below the function entry statement. For example:

```
ACS .5                  (Radians)
  1.Ø471975512
SET DEG
ACS .5                  (Degrees)
  6Ø
SET GRAD
ACS .5                  (Grads)
  66.6666666667
```

Program Execution

The ACS function returns a numeric result and can be specified as part of a numeric expression. The following examples illustrate several ways the ACS function can be specified in a numeric expression and evaluated under program control:

```
100 C8=ACS (X↑2)
110 PRINT @17:3*A+ACS(B)/7
120 M4=360/ACS(M3)
```

The keyword ACS can be entered as ACOS, however, the BASIC interpreter always converts ACOS to ACS in a program listing.

The ACS parameter must be specified in the range −1 through +1 or an error occurs and program execution is aborted.

The angle returned by the ACS function is always between 0° and 180°.

THE ASN FUNCTION

Syntax Form:

{ASIN / ASN} numeric expression

Purpose

The ASN (Arc Sine) function reduces the specified numeric expression to a numeric constant, interprets the numeric constant as the sine of an angle, and returns the angle expressed in the current trigonometric units for the system.

Explanation

ASN is the Opposite of SIN

The ASN function is the opposite of the SIN function. The SIN function requires an angle for a parameter and returns the sine of the angle. The ASN function, on the other hand, requires the sine of an angle for a parameter and returns the corresponding angle expressed in the current trigonometric units for the system. For example, the sine of 34 degrees is .529; the arc sine of .529 is 34 degrees. The ASN parameter must be specified within the range −1 through +1 or an error occurs and the appropriate error message is printed on the GS display.

Immediate Execution

The BASIC interpreter immediately returns the angle of any value between −1 and +1, if the ASN function and the value are entered directly from the GS keyboard and the RETURN key is pressed. The BASIC interpreter assumes the value is the sine of an angle measured in radians unless the system environment is set to degrees or grads. (Refer to the SET statement in the Environmental Control section of this manual for details.) The angle is returned to the GS display and printed one line below the function entry statement. For example:

```
ASN .5                  (Radians)
  Ø.523598775598
SET DEG
ASN .5                  (Degrees)
  3Ø
```

```
SET GRAD
ASN .5                (Grads)
  33.3333333333
```

Program Execution

The ASN function returns a numeric result and can be specified as part of a numeric expression. The following examples illustrate several ways the ASN function can be specified in a numeric expression and evaluated under program control:

```
500 X=ASN(Y13)
510 DRAW 60*ASN(X),180*ASN(Y)
520 M2=M2+ASN(F5)
```

The keyword ASN can be entered as ASIN, however, the BASIC interpreter always converts ASIN to ASN in a program listing.

The ACS parameter must be specified in the range −1 through +1 or an error occurs and program execution is aborted.

The angle returned by the ASN function is always between −90° and +90°.

THE ATN FUNCTION

Syntax Form:

{ATAN / ATN} numeric expression

Purpose

The ATN (Arc Tangent) function reduces the specified numeric expression to a numeric constant, interprets the numeric constant as the tangent of an angle, and returns the angle expressed in the current trigonometric unit for the system.

Explanation

ATN is the Opposite of TAN

The ATN function is the opposite of the TAN function. The TAN function requires an angle as a parameter and returns the tangent of the angle. The ATN function, on the other hand, requires the tangent of an angle and returns the corresponding angle expressed in the current trigonometric units for the system. For example, the tangent of 34 degrees is .675; the arc tangent of .675 is 34 degrees.

Immediate Execution

The BASIC interpreter immediately returns the angle of any tangent when the ATN function and the tangent are entered directly from the GS keyboard and the RETURN key is pressed. The BASIC interpreter assumes the ATN parameter is the tangent of an angle measured in radians unless the system environment is set to degrees or grads. (Refer to the SET statement in the Environmental Control section of this manual for details.) The minimum angle returned is −90 degrees and the maximum angle returned is +90 degrees, if the system environment is set to degrees. For example:

```
ATN 2                          (Radians)
  1.1Ø714871779
SET DEG
ATN 2                          (Degrees)
  63.4349488229
SET GRAD
ATN 2                          (Grads)
  7Ø.4832764699
```

Program Execution

The ATN function returns a numeric result and can be specified as part of a numeric expression. The following examples illustrate several ways the ATN function can be specified in a numeric expression and evaluated under program control:

```
2500 VIEWPORT ATN (X1),ATN(X2)+1,ATN(Y1),ATN(Y2)+1
2510 N=Q/ATN(.456)
2520 PRINT @19:"The Rotation Angles is ";ATN(R3)+30
```

The keyword ATN can be entered as ATAN, however, the BASIC interpreter always converts ATAN to ATN in a program listing.

The angle returned by the ATN function is always between $-90°$ and $+90°$.

THE COS FUNCTION

Syntax Form:

COS numeric expression

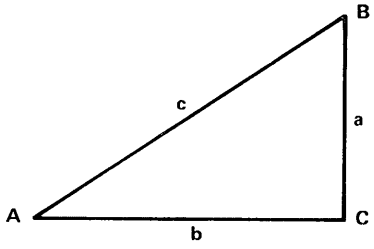
PURPOSE

The COS (Cosine) function reduces the specified numeric expression to a numeric constant, interprets the numeric constant as an angle, and returns the cosine of the angle expressed in the current trigonometric units for the system.

EXPLANATION

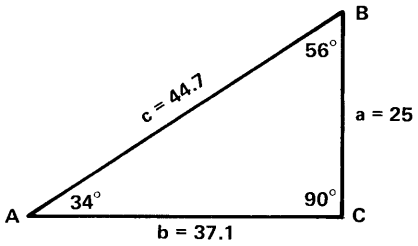
Trigonometry Review

The cosine of an acute angle in a right triangle is the ratio between the side adjacent to the acute angle and the hypotenuse. For example, given any right triangle ABC:



$$\text{COS A} = \frac{\text{adjacent side}}{\text{hypotenuse}} = \frac{b}{c}$$

A practical example:



COS 34° = 37.1/44.7 = .829

NOTE: All values are approximate due to rounding.

Immediate Execution

The BASIC interpreter returns the cosine of any angle immediately after the COS function is entered and the RETURN key is pressed. The BASIC interpreter assumes the angle is specified in radians unless the system environment is set to degrees or grads. For example:

```
COS 34                  (Radians)
  -Ø.848570274785
SET DEG
COS 34                  (Degrees)
  Ø.829037572555
SET GRAD
COS 34                  (Grads)
  Ø.860742027ØØ4
```

Refer to the Environmental Control section for an explantion of the SET statement.

Program Execution

Because the COS function returns a numeric result, the function can be specified as part of a numeric expression. The following examples illustrate how the COS function can be specified in a numeric expression and evaluated under program control:

```
5ØØ F7=COS(X+Y)-1
51Ø G=-5*COS(X*Y+5Ø)
52Ø WRITE @16:1+COS(C-D),1-COS(C+D)
53Ø MOVE 65+SIN(X),5Ø+COS(Y)
```

Parameter Limits

The COS parameter must be in the range ± 4.116E+5 radians or a SIZE error occurs and Ø is returned as the result of the function.

THE DEF FN STATEMENT

Syntax Form:

Line number DEF FN letter (numeric variable) = numeric expression

Descriptive Form:

Line number DEF FN any letter (numeric variable) = function to be defined

Purpose

The DEF FN (Define Function) statement allows a numeric expression to be defined and executed as a function.

Explanation

If a numeric expression is defined and evaluated several times throughout a program, it is convenient to define the numeric expression as a function and then specify the function symbol instead of the numeric expression. A maximum of 26 numeric expressions can be defined in this way (FNA through FNZ). Each numeric expression is limited to one numeric variable. For example:

```
.
.
.
300 DEF FNA (X) = X↑2+2*X+4
310 READ Y
320 PRINT FNA(Y)
330 D = SQR (FNA(5))+3
.
.
.
```

In line 300, the numeric expression X↑2+2*X+4 is defined as FNA (function A). Notice that the numeric variable specified in the numeric expression is also specified in parentheses after the letters FNA.

Line 31Ø causes the BASIC interpreter to assign a value to Y from the DATA statement located elsewhere in the program. Line 32Ø evaluates the function FNA using the currently assigned value of Y and prints the result on the display. If 3 is assigned to Y in line 31Ø, for example, then the BASIC interpreter evaluates the numeric expression X↑2+2*X+4 replacing X with the value 3 and returns the result (19) to the GS display; if Y=4 then the BASIC Interpreter returns the value (28) to the display.

In line 33Ø the numeric expression SQR (FNA(5))+3 is reduced to a numeric constant and assigned to the variable D. The function of A is evaluated first. Since the value 5 is specified in parentheses, the function of A is evaluated using 5 for the value of X. (This has no effect on the assigned value of X in other parts of the program.) The BASIC interpreter reduces FNA(5) to 39, takes the square root of 39 (6.2449979984), adds 3 and assigns the result (9.2449979984) to the numeric variable D. When this function is specified in other program statements, it is specified in the form FNA (numeric expression). The numeric expression is reduced to a numeric constant and interpreted as the value of X. It can be seen from this example that it is much easier to specify FNA(5) than it is to specify X↑2+2*X+4.

A DEF FN statement must be executed for each defined function in the program before the function is used in another statement.

THE DET FUNCTION

Syntax Form:

[Line number] array variable = INV array variable
[[Line number] numeric variable =] DET

NOTE

As indicated in the syntax form above, the INV (Inverse) function must be performed in order to use the DET function. The value of the determinant is computed during the INV process. For this reason, it is important to understand the INV function before attempting to use DET.

In order to perform the DET function, a 4051 Graphic System must be equipped with a Matrix Functions ROM pack.

Purpose

The DET (Determinant) function returns the value of the determinant.

Explanation

The DET function returns the determinant of the square part of a matrix which has been used as a parameter for the INV function. For example, if A and B are 2x3 matrices:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 2 & -3 \end{bmatrix}, \quad \mathbf{B = INV(A)} = \begin{bmatrix} 2 & -1 & 3 \\ -1 & 1 & -3 \end{bmatrix}, \quad \mathbf{DET = 1}$$

As indicated above, after the INV function is performed on matrix A, the DET function is used to compute the determinant of the square part (the shaded portion) of A.

The DET function is always performed after the matrix has been supplied to the INV function: in other words, the INV function is performed first, then the DET function.

As an example, the following sequence of statements may be used to find the determinant of a 5x5 matrix:

```
DEL A,B
DIM A(5,5),B(5,5)
INPUT A
B = INV(A)
DET
```

Notice that the statement B = INV(A) is executed before the statement DET.

The DET function does not necessarily have to be performed **immediately** after the INV function. No matter how many BASIC statements have been executed since the last use of the INV function, the DET function always returns the determinant of the square part of the matrix most recently supplied to the INV function.

When executing statements directly from the keyboard, the value of the determinant may be obtained by entering the DET and pressing the RETURN key.

When the DET function appears in a BASIC program, the value of the determinant may be obtained from a statement such as 100 PRINT DET. (A statement such as 100 DET results in an error.)

It is important to keep in mind that the result of the DET function is a numeric constant, and should be assigned to a numeric variable (that is, a variable which has not previously appeared in a DIM statement). Thus, the target variable should not have the same name as an array. For example, if the INV function is performed in a 3x3 matrix named A, the assignment A = DET replaces all nine elements of A by the numeric constant resulting from the DET function.

NOTE

The DET function always returns the determinant of the square part of the matrix most recently supplied to the INV function. Forgetting to perform the INV function before the DET function may result in an incorrect answer. For example, when finding the determinant of a matrix called B, if DET is performed before INV(B), the answer which appears on the display is not the determinant of matrix B; it is the determinant of whatever matrix was last supplied to the INV function.

THE EXP FUNCTION

Syntax Form:

EXP numeric expression

Purpose

The EXP (e to the power) function raises the base e (natural logarithm base) to the power specified by the numeric expression.

Explanation

Immediate Execution

If the EXP function and its parameter are entered directly from the GS keyboard and the RETURN key is pressed, the BASIC interpreter immediately returns the base e raised to the specified power. For example:

```
EXP 2
  7.389Ø56Ø9893
```

When EXP 2 is entered from the GS keyboard and the RETURN key is pressed, the result (7.389Ø56Ø9893) is returned to the GS display and printed one line below the entry statement as shown above.

Program Execution

The EXP function returns a numeric result and can be specified as part of a numeric expression. The following examples show several ways the EXP function can be specified in a numeric expression and evaluated under program control:

```
14Ø S=EXP(4)+5/8
15Ø Y=X*EXP(X+2)
```

THE IDN ROUTINE

Syntax Form:

[Line number] CALL { "IDN" / string variable }, array variable

Descriptive Form:

[Line number] CALL routine call name , target variable

NOTE

In order to perform the IDN routine, a 4051 Graphic System must be equipped with a Matrix Functions ROM Pack.

PURPOSE

The IDN (Identity) routine creates a matrix whose elements are 1's along the diagonal, and 0's elsewhere.

EXPLANATION

The IDN routine generates a matrix whose elements are 1's along the diagonal, and 0's elsewhere. For example, the IDN routine may be used to generate the following 3x3 matrix:

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

I(1,1),I(2,2), and I(3,3) are the diagonal elements of matrix I, and have been assigned the value 1. All other elements are 0's.

When the IDN routine is used to generate a square matrix as in the above example, the result is called an identity matrix. In this example, matrix I is the 3x3 identity matrix.

The result of the IDN function is assigned to an array variable, called the target variable. The result of the IDN function always has the same dimensions as the target variable. For instance, it I is to be the target for the result of the IDN routine, and is dimensioned in a DIM statement to be a 5x5 matrix, performing the IDN routine results in the 5x5 identity matrix shown below:

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

The IDN Routine and Non-Square Matrices

A matrix does not have to have a square shape to be used as the target for the IDN routine. The IDN routine assigns values to the elements of a non-square array I, just as it did to the square matrices in the preceding examples: diagonal elements (elements I(1,1),I(2,2),...,I(K,K)) are assigned the value 1, and all other elements are assigned the value 0. When the target variable is dimensioned to be a non-square matrix, however, the IDN routine produces a matrix having at least one row or column filled with 0's. The rows or columns of 0's are the ones which lie outside the square part of the matrix. (The square part of a matrix is the largest square portion of the matrix which includes the upper-left corner.)

For example, when I is dimensioned to be a 3x5 array, performing the IDN function results in the following matrix:

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

Note that the diagonal elements I(1,1),I(2,2), and I(3,3), are all 1's, and the rest of the elements are 0's. Also, the last two columns of the matrix have 0's assigned to every element.

When the target variable has a non-square shape, the result of the IDN routine resembles an identity matrix (the square part is an identity matrix). But because of the extra rows or columns of 0's, the non-square matrix does not have the properties of an identity matrix. For an explanation of the special properties of identity matrices, refer to the topic "Properties of Identity Matrices" on the following pages.

Dimensioning the Target Variable

Before performing the IDN routine, a target variable must be dimensioned in a DIM statement, using two subscripts to indicate the number of rows and columns in the result matrix. For instance, if variable A is to receive the result of the IDN routine, and the function is being used to generate the 3x3 identity matrix, A must be dimensioned as follows:

```
DIM A(3,3)
```

Using only one subscript to dimension the target variable causes an error to occur when the IDN routine is performed. Any valid numeric variable name may be used as the target for the result of the IDN routine. An array can serve as the target for the IDN routine without having numeric values assigned to each element.

PROPERTIES OF IDENTITY MATRICES

Multiplying by an Identity Matrix

When a square matrix is multiplied by the identity matrix of the same size, the result is the original matrix. That is, the following is true for any square matrix A:

A MPY I = I MPY A = A

In this expression, MPY is matrix multiplication and I is the identity matrix having the same dimensions as A.

For example, let A be a 2x2 matrix and I be the 2x2 identity matrix as shown below:

$$A = \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix} \quad , \quad I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Multiplying A by I (or I by A) results in the same matrix A.

$$A \text{ MPY } I = \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix} = A$$

$$I \text{ MPY } A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix} = A$$

(Refer to the MPY function for how to perform these calculations.)

The above rule only holds when A is square and I is the identity matrix of the same size as A. If I is a non-square matrix generated by performing the IDN routine on a non-square target variable I, then it is not true that A MPY I = I MPY A = A. Instead, the product of A and I (or of I and A) may return a matrix which resembles A, but has chopped off part of A, or added rows or columns of 0's to A.

When an Identity Matrix is the Result of Performing Matrix Multiplication

One matrix is the inverse of another matrix if their product results in an identity matrix. In other words, matrix B is the inverse of matrix A if the following is true:

A MPY B = B MPY A = I

In this expression, MPY is matrix multiplication and I is the identity matrix having the same dimensions as A and B. A and B must both be square matrices.

As an example, let A and B be the 2x2 matrices shown below:

$$A = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}, \quad B = \begin{bmatrix} -2 & 3 \\ 3 & -4 \end{bmatrix}$$

The products A MPY B and B MPY A both result in the 2x2 identity matrix. This can be verified by performing the MPY function on A and B:

$$C = A \text{ MPY } B = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} -2 & 3 \\ 3 & -4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$D = B \text{ MPY } A = \begin{bmatrix} -2 & 3 \\ 3 & -4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

(Refer to MPY for an explanation of how to compute these products.) Since both A MPY B and B MPY A result in the 2x2 identity matrix, A is the inverse of B.

THE INT FUNCTION

Syntax Form:

INT numeric expression

Purpose

The INT (Integer) function reduces the specified numeric expression to a numeric constant and returns the largest integer possible without exceeding the value of the numeric constant.

Explanation

Immediate Execution

If the INT function and a numeric expression are entered into the system directly from the GS keyboard, the BASIC interpreter immediately evaluates the numeric expression and returns the largest integer possible without exceeding the value of the numeric expression. For example:

```
INT (1−4.33)
  −4
```

In this example, the numeric expression (1−4.33) is evaluated to −3.33. The BASIC interpreter returns a −4, because −4 is the largest integer value that doesn't exceed −3.33.

Program Execution

Because the INT function returns a numeric value, the INT function can be specified as part of a numeric expression. The following examples show several ways the INT function can be specified in a numeric expression and evaluated under program control:

```
300 LET J=INT(S3)+3
310 IF INT(H)=0 THEN 1205
320 WRITE INT(A),INT(B),INT(C)/10* 87
```

Parentheses are not normally required around the INT parameter at statement entry time; however, the BASIC interpreter places parentheses around the parameter in a program listing for clarity.

NOTE

Because of the binary nature of the 4050 Series Graphic System, even though 1/3 + 2/3 = 1, INT (1/3 + 2/3) = 0. In such a case, use: INT (X + (INT (X) = X − 1)).

THE INV FUNCTION

Syntax Form:

[Line number] array variable = INV array variable

Descriptive Form:

[Line number] target variable = INV parameter variable

NOTE

In order to perform the INV function, a 4051 Graphic System must be equipped with a Matrix Functions ROM Pack.

Purpose

The INV (Inverse) function performs matrix inversion and solves systems of linear equations.

Explanation

The INV function returns a new matrix which has the same size and shape as the original array. The square part of the new matrix is the inverse of the square part of the original matrix, and any columns which lie outside the square part in the new matrix, represent solutions to sets of linear equations. The square part of a matrix is the largest square that can be blocked off, starting in the upper left-hand corner of the matrix. A detailed explanation of what this means is given later. Meanwhile, an example is given to illustrate what the INV function does:

$$A = \begin{bmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 0 \end{bmatrix} \quad , \quad B = INV(A) = \begin{bmatrix} 2 & -3 & -7 & 8 \\ -1 & 2 & 5 & -4 \end{bmatrix}$$

Notice that when the 2x4 matrix A is supplied to the INV function, the result B = INV(A) is also a 2x4 matrix.

Solving Equations

In the previous example, both matrices A and B have two columns which lie outside the square part. The extra columns are shown below:

$$A = \begin{bmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 0 \end{bmatrix}, \quad B = INV(A) = \begin{bmatrix} 2 & -3 & -7 & 8 \\ -1 & 2 & 5 & -4 \end{bmatrix}$$

The two extra columns in B represent solutions to two different sets of linear equations. The first set of equations is

$$\begin{aligned} 2x + 3y &= 1 \\ x + 2y &= 3 \end{aligned}$$

and the first extra column in B represents the solution $x = -7$ and $y = 5$. The second set of equations is

$$\begin{aligned} 2x + 3y &= 4 \\ x + 2y &= 0 \end{aligned}$$

and the second extra column in B represents the solution $x = 8$ and $y = -4$.

CAUTION

It may be tempting in this example to think of matrix A as representing the following system of linear equations:

$$\begin{aligned} 2x + 3y + z &= 4 \\ x + 2y + 3z &= 0 \end{aligned}$$

This is not a correct interpretation. The INV function always solves systems of N equations in N unknowns, where N stands for the number of rows in the parameter matrix. Since there are two rows in matrix A above, INV(A) provides solutions to systems of two equations in two unknowns (x and y).

When the INV function is performed on a square matrix, the result is the inverse of the original matrix. The INV function can only be successfully performed when the square part of the parameter matrix has an inverse. Not every square matrix has an inverse. For instance, the matrix C shown below does not have an inverse:

$$C = \begin{bmatrix} 2 & -2 \\ -5 & 5 \end{bmatrix}$$

When the INV function is unable to compute an answer because the matrix has no inverse, the system of linear equations has no solution (or does not have a unique solution). This can be treated as a SIZE error.

There are no extra columns to the right of the square part, however, so the result does not provide solutions to sets of linear equations.

Dimensioning the Array

When the INV function is performed, a new matrix is generated which has the same size and shape as the original matrix. The new matrix must be assigned to a target variable. Both arrays must be dimensioned in a DIM statement, using two subscripts to indicate the number of rows and columns. Since the new matrix has the same size and shape as the original matrix, the subscripts used to dimension the target variable must be the same as those used to dimension the parameter variable. The target variable may have the same name as the parameter variable. However, when the statement is executed, the **original matrix is replaced by the new matrix** generated by the INV function.

Not assigning a numeric value to every element of the parameter matrix causes an UNDEFINED VARIABLE error to occur when the INV function is performed.

Attempting to perform the INV function on a matrix which has more rows than columns, causes a SHAPE ERROR: the parameter matrix must have at least as many columns as rows. Attempting to perform the INV function on a matrix which has more than 255 rows or more than 255 columns also causes a SHAPE ERROR.

THE LGT FUNCTION

Syntax Form:

LGT numeric expression

Purpose

The LGT (Logarithm Base 1Ø) function reduces the specified numeric expression to a numeric constant and returns the logarithm of the numeric constant to the base 1Ø.

Explanation

The LGT function returns the logarithm of the numeric expression to the base 1Ø. For example, LGT 7 (the logarithm of 7) is equal to Ø.845Ø98Ø4ØØ14. This means that 7 is equal to 1Ø raised to the Ø.845Ø98Ø4ØØ14th power, or

$$7 = 1Ø^{0.845098040014}$$

Immediate Execution

If the LGT function is entered directly from the GS keyboard and the RETURN key is pressed, the BASIC interpreter immediately returns the result to the display as shown below:

```
LGT 7
  Ø.845Ø98Ø4ØØ14
```

Program Execution

Because the LGT function returns a numeric result, the function can be specified as part of a numeric expression. For example:

```
21Ø LET A = LGT(B) + LGT(C) - LGT(D)
22Ø IF LGT(V8)>6 THEN 8ØØ
23Ø RDRAW 1ØØ*LGT(X),1ØØ*LGT(Y)
```

THE LOG FUNCTION

Syntax Form:

LOG numeric expression

Purpose

The LOG (Logarithm Base e) function reduces the specified numeric expression to a numeric constant and returns the logarithm of the numeric constant to the base e.

Explanation

The LOG function returns the natural logarithm of the numeric expression. For example, LOG 5 (the natural logarithm of 5) is equal to 1.6Ø943791243. This means that 5 is equal to the base e (2.71828182846) raised to the 1.6Ø94379124th power, or

$$5 = e^{1.6Ø943791243}$$

Immediate Execution

If the LOG function is entered directly from the GS keyboard and the RETURN key is pressed, the BASIC interpreter immediately returns the result to the GS display as shown below:

```
LOG 5
  1.6Ø943791243
```

Program Execution

Because the LOG function returns a numeric result, it can be specified as part of a numeric expression. For example:

```
77Ø H = LOG(89)+1
78Ø RMOVE 1Ø*LOG(X),1Ø*LOG(Y)
79Ø PRINT "The natural logarithm of Y is "; LOG(Y)
```

THE MPY FUNCTION

Syntax Form:

[Line number] array variable = array variable MPY array variable

Descriptive Form:

[Line number] target variable = parameter variable MPY parameter variable

NOTE

In order to perform the MPY function, a 4051 Graphic System must be equipped with a Matrix Functions ROM Pack.

Purpose

The MPY (Matrix Multiplication) function returns the matrix product of two arrays.

Explanation

When the MPY function is performed, the result is a new matrix. For example:

$$A = \begin{bmatrix} 1 & 2 \\ 0 & -3 \end{bmatrix}, \quad B = \begin{bmatrix} 2 & 8 \\ 3 & 4 \end{bmatrix}$$

$$C = A \text{ MPY } B = \begin{bmatrix} 1 & 2 \\ 0 & -3 \end{bmatrix} \begin{bmatrix} 2 & 8 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 8 & 16 \\ -9 & -12 \end{bmatrix}$$

The elements of the new matrix C are found by multiplying each element of the Ith **row** of the **first** matrix by the corresponding element of the Jth **column** of the **second** matrix, then adding the values. The sum is the I,Jth element of the product matrix.

The MPY function can only be performed when the number of columns in the first matrix is equal to the number of rows in the second matrix. For instance, A MPY B can be performed for the following matrices:

$$A = \begin{bmatrix} 2 & 3 & -4 \\ 3 & -1 & 2 \end{bmatrix} \quad , \quad B = \begin{bmatrix} -1 & 3 & 0 \\ 2 & 0 & 5 \\ 0 & 4 & 1 \end{bmatrix}$$

2x3 3x3

As indicated above, matrix A has three columns and matrix B has three rows. Since the number of columns in A is the same as the number of rows in B, the product C = A MPY B can be computed.

All elements of A and B must be defined.

The result of A MPY B is a new matrix which has as many rows as A, and as many columns as B. For example:

$$A = \begin{bmatrix} 3 & 1 \\ -1 & 1 \\ 7 & 0 \end{bmatrix} \quad , \quad B = \begin{bmatrix} 2 & -1 & 0 \\ 1 & 5 & -2 \end{bmatrix}$$

$$C = A \text{ MPY } B = \begin{bmatrix} 3 & 1 \\ -1 & 1 \\ 7 & 0 \end{bmatrix} \begin{bmatrix} 2 & -1 & 0 \\ 1 & 5 & -2 \end{bmatrix} = \begin{bmatrix} 7 & 2 & -2 \\ -1 & 6 & -2 \\ 14 & -7 & 0 \end{bmatrix}$$

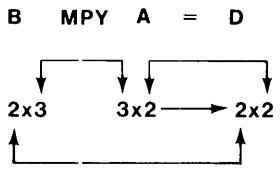
$$D = B \text{ MPY } A = \begin{bmatrix} 2 & -1 & 0 \\ 1 & 5 & -2 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ -1 & 1 \\ 7 & 0 \end{bmatrix} = \begin{bmatrix} 7 & 1 \\ -16 & 6 \end{bmatrix}$$

Since matrix A has two columns and matrix B has two rows, C = A MPY B can be computed. The result has as many rows as A and as many columns as B, which means that C is a 3x3 matrix:

A MPY B = C

3x2 2x3 → 3x3

Since B has three columns and A has three rows, the product D = B MPY A can also be computed. The result has as many rows as B and as many columns as B, which means D is a 2x2 matrix.



Notice that both A MPY B and B MPY A can be performed, but the resulting matrices are not the same, and are not even of the same size.

Dimensioning the Arrays

When the MPY function is performed, a new matrix is generated. The result must be assigned to a target variable. The target variable and parameter variable must be dimensioned in a DIM statement, using two subscripts to indicate the number of rows and columns in the resulting matrix. In order to perform C = A MPY B, the first subscript used to dimension C must be the same as the first subscript used to dimension A, and the second subscript used to dimension C must be the same as the second subscript used to dimension B. For example:

```
100 DIM A(7,3),B(3,12),C(7,12)
```

The target variable must not have the same name as either of the parameters.

THE PI FUNCTION

Syntax Form:

PI

Purpose

The PI function returns the numeric value 3.14159265359.

Explanation

Immediate Execution

If the letters PI are entered directly from the GS keyboard and the RETURN key is pressed, the BASIC interpreter immediately returns the value 3.14159265359. The value of PI can also be entered as part of a numeric expression and evaluated immediately. For example:

```
2*PI*3/4
  4.71238898Ø038
```

In this example, the numeric expression 2*PI*3/4 is entered from the GS keyboard and the RETURN key is pressed. The result (4.71238898Ø038) is returned to the GS display and printed on the next line.

Program Execution

Because the PI function returns a numeric value, the function can be placed in a numeric expression and evaluated under program control. For example:

```
3ØØ C=2*PI*R
31Ø DRAW 65+SIN(2*PI),5Ø+COS(2*PI)
32Ø PRINT @33:PI;SIN(PI);COS(PI)
```

THE RND FUNCTION

Syntax Form:

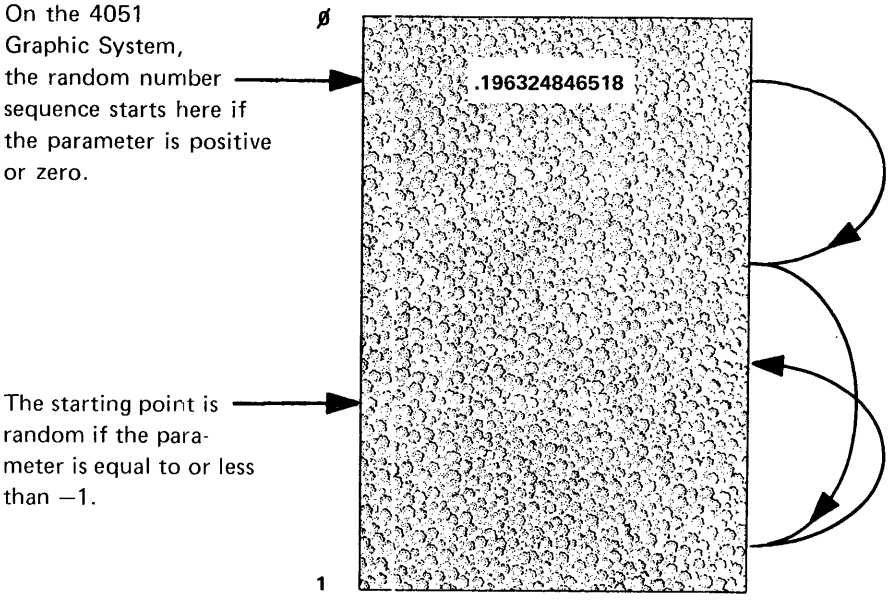
RND numeric expression

Purpose

The RND (Random Number) function returns a random number between Ø and 1.

Explanation

The RND function causes an internal pseudorandom number generator to output a random number from Ø through 1. A model of this generator is shown below.



RANDOM NUMBER GENERATOR

The random number generator contains approximately 140 trillion numbers within the range Ø to 1. These numbers are linked together in a chain in such a way as to appear random when they are output from the generator. Each time the RND function is executed, one number in the chain is output. The next time the RND function is executed, another number in the chain is output, and so on.

The parameter of the RND function determines how the random number chain is used, in the following manner:

N	RND(N)
$-1<N\leq 0$	A fixed starting place in the chain is returned; entering −.66 will always return the same number (on the 4051 Graphic System, RND (−.66) = .500108400217).
$N>0$	Return the next number in the chain; if no location in the chain has been established, the first number in the chain is returned (RND(0) has been arbitrarily chosen as the first number in the chain).
$N\leq -1$	A random starting place in the chain is returned.

The best method for using the RND function is to select a random starting point in the chain (RND(−1)), and from then on use RND(1) to progress through the chain from that point.

The values for the first numbers in the chain have been arbitrarily set as follows:

System	RND(0)
4051	0.196324846510
4052	0.706280095237
4054	0.88093139039

In this way a program can determine internally what kind of Graphic System it is running on.

THE SGN FUNCTION

Syntax Form:

{ SIGN }
{ SGN } numeric expression

Purpose

The SGN (Signum) function reduces the specified numeric expression to a numeric constant and returns a 1 if the numeric constant is positive, a −1 if the numeric constant is negative, and a Ø if the numeric constant is zero.

Explanation

The SGN function is normally used to find the sign of a number; either positive, negative, or zero. For example:

```
23Ø IF SGN(X)= −1 THEN 67Ø
```

This statement uses the SGN function as the basis for a conditional branch. If the assigned value of X is negative, then the SGN function returns a −1 and program execution branches to line 67Ø. If the value of X is not negative, then the program continues executing in sequence.

The SGN function returns a numeric result and can be specified as part of a numeric expression. For example:

```
56Ø LET Z8=SGN(Y)*SIN(3*PI/2)
57Ø PRINT @15: SGN(9+4X−X↑2)
58Ø DRAW 65+4Ø*SGN(X),5Ø+4Ø*SGN(Y)
```

NOTE

The SGN function only returns a Ø when the parameter is exactly Ø, and does not use the FUZZ comparison. If a "fuzzy" SGN function is needed, use: (SGN (X(X=0)).*

THE SIN FUNCTION

Syntax Form:

SIN numeric expression

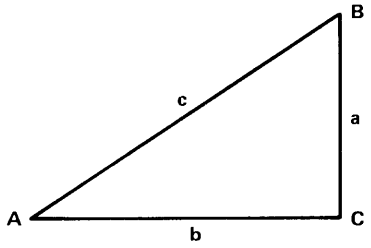
Purpose

The SIN (Sine) function reduces the specified numeric expression to a numeric constant, interprets the numeric constant as an angle, and returns the sine of the angle expressed in the current trigonometric units for the system.

Explanation

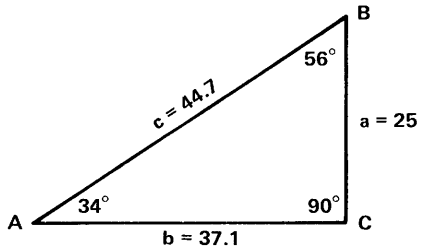
Trigonometry Review

The sine of an acute angle within a right triangle is the ratio between the side opposite the acute angle and the hypotenuse. Given any right triangle ABC:



$$\text{SIN A} = \frac{\text{opposite side}}{\text{hypotenuse}} = \frac{a}{c}$$

A practical example:



SIN 34° = 25/44.7 = .559

NOTE: All values are approximate due to rounding.

Immediate Execution

The BASIC interpreter returns the sine of any angle immediately after the SIN function is entered and the RETURN key is pressed. The BASIC interpreter assumes the angle is specified in radians unless the system environment is set to degrees or grads. For example:

```
SIN 34                          (Radians)
  Ø.52908268612
SET DEG
SIN 34                          (Degrees)
  Ø.559192903471
SET GRAD
SIN 34                          (Grads)
  Ø.5Ø9Ø4141575
```

Refer to the Environmental Control section for an explanation of the SET statement.

Program Execution

Because the SIN function returns a numeric result, the function can be specified as part of a numeric expression. The following examples illustrate how the SIN function can be specified in a numeric expression and evaluated under program control:

```
1ØØ A=SIN(X)+3
11Ø LET P4=2*SIN(A+B)*COS(A-B)/3
12Ø LET P6=SIN(2*PI/3)/6
13Ø DRAW SIN(A-B)↑2,SIN(A+B)↑4
```

Parameter Limits

The SIN parameter must be in the range ± 4.116E+5 radians or a SIZE error occurs and Ø is returned as the result of the function.

THE SQR FUNCTION

Syntax Form:

SQR numeric expression

Purpose

The SQR (Square Root) function reduces the specified numeric expression to a numeric constant and returns the square root of the numeric constant.

Explanation

Immediate Execution

If the SQR function is entered directly from the GS keyboard and RETURN key is pressed, the BASIC interpreter immediately returns the square root of the specified numeric expression. For example:

```
SQR (24+15)
  6.2449979984
```

Program Execution

Because the SQR function returns a numeric result, the SQR function can be specified as part of a numeric expression and evaluated under program control. For example:

```
29Ø LET K7=SQR (A+B/2)
3ØØ C=SQR(A↑2 + B↑2)
31Ø PRINT "The square root of X is "; SQR(X)
```

Specifying a Negative Parameter

If the parameter of the SQR function is negative, the SQR function returns the positive square root of the parameter and generates a SIZE error condition. This error is treated as a fatal error and program execution is aborted, unless an ON SIZE THEN... statement has been previously executed in the BASIC program. (Refer to the Handling Interrupts section for details on how to handle SIZE errors.)

THE SUM FUNCTION

Syntax Form:

SUM array variable

Purpose

The SUM function returns the algebraic sum of the elements in the specified array.

Explanation

Immediate Execution

If the SUM function and its parameter are entered directly into the system from the GS keyboard and the RETURN key is pressed, the BASIC interpreter returns the algebraic sum of the array elements to the GS display. For example:

```
SUM X
 1Ø
```

If X is an array with the elements 1,2,3,4, then entering SUM X from the GS keyboard and pressing the RETURN key returns the algebraic sum of the elements in X. The sum is printed on the GS display one line below the entry statement. The array X can be a one dimensional array or a two dimensional array; it doesn't matter.

Program Execution

Because the SUM function returns a numeric result, the function can be specified as part of a numeric expression. The following examples illustrate several ways the SUM function can be specified in a numeric expression and evaluated under program control:

```
87Ø C=SUM(A)+SUM(B)
88Ø SCALE SUM(X)+3,SUM(Y)-3
```

Parentheses are not normally required around the SUM parameter at statement entry time; however, the BASIC interpreter places parentheses around the parameter in a program listing for clarity.

THE TAN FUNCTION

Syntax Form:

TAN numeric expression

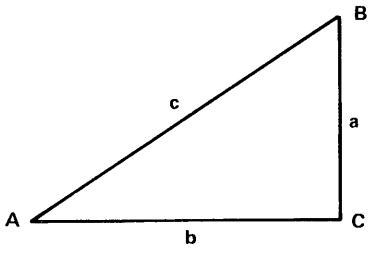
Purpose

The TAN (Tangent) function reduces the specified numeric expression to a numeric constant, interprets the numeric constant as an angle, and returns the tangent of the angle expressed in the current trigonometric units for the system.

Explanation

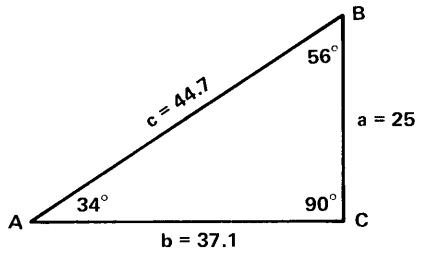
Trigonometry Review

The tangent of an acute angle in a right triangle is the ratio between the side opposite the acute angle and the side adjacent to the acute angle. For example, given any right triangle ABC:



$$\text{TAN A} = \frac{\text{opposite side}}{\text{adjacent side}} = \frac{a}{b}$$

A practical example:



TAN 34° = 25/37.1 = .675

NOTE: All values are approximate due to rounding.

Immediate Execution

The BASIC interpreter returns the tangent of any angle immediately after the TAN function is entered and the RETURN key is pressed. The BASIC interpreter assumes the angle is specified in radians unless the system environment is set to degrees or grads. For example:

```
TAN 34                    (Radians)
  -Ø.623498962716
SET DEG
TAN 34                    (Degrees)
  Ø.674508516842
SET GRAD
TAN 34                    (Grads)
  Ø.591398351399
```

Refer to the Environmental Control section for a complete explanation of the SET statement.

Program Execution

Because the TAN function returns a numeric result, the function can be specified as part of a numeric expression. The following examples illustrate how the TAN function is specified in a numeric expression and evaluated under program control:

```
1ØØØ Q = TAN(Y)+1
1Ø1Ø MOVE TAN(X)*65,TAN(Y)*50
1Ø2Ø R5 = 2/TAN(B9-5)
1Ø3Ø PRINT @33: 1/SQR(TAN(X))
```

Parameter Limits

The TAN parameter must be in the range ±4.116E+ radians or a SIZE error occurs and Ø is returned as the result of the function. If the parameter of the TAN function is specified as +9Ø degrees or -9Ø degrees, the largest number possible is returned and an error is not generated.

THE TRN FUNCTION

Syntax Form:

[Line number] array variable = TRN array variable

Descriptive Form:

[Line number] target variable = TRN parameter variable

NOTE

In order to perform the matrix function, a 4051 Graphic System must be equipped with a Matrix Functions ROM Pack.

Purpose

The TRN (Transpose) function returns the transpose of a matrix.

Explanation

When the TRN function is performed on a matrix, the result is a new matrix found by making the columns into rows, or the rows into columns. For instance, when B = TRN(A) is performed, each row in A becomes the corresponding column in B. This is equivalent to saying that each **column** in A becomes the corresponding **row** in B. For example:

$$A = \begin{bmatrix} 1 & 3 & 5 \\ 8 & -2 & 0 \end{bmatrix}, \qquad B = TRN(A) = \begin{bmatrix} 1 & 8 \\ 3 & -2 \\ 5 & 0 \end{bmatrix}$$

In this example, A is a 2x3 matrix. When the TRN function is performed on A, the result is the 3x2 matrix B shown above.

Notice that the first row of A is the same as the first column of B, and the second row of A is the same as the second column of B. Likewise, the first **column** of A is the same as the first **row** of B, the second column of A is the same as the second row of B, and so on. Matrix B is the transpose of matrix A.

When the TRN function is performed, the number of rows and columns are reversed. In the above example, A is a 2x3 matrix, so B=TRN(A) is a 3x2 matrix. In general terms, when A is a matrix having K rows and N columns, B=TRN(A) is a matrix having N rows and K columns.

If A is a matrix consisting of one row, the TRN function returns a matrix consisting of one column, and vice versa. For instance:

$$\begin{bmatrix} A = & 6 & 1 & 8 \end{bmatrix} \quad , \qquad B = TRN(A) = \begin{bmatrix} 6 \\ 1 \\ 8 \end{bmatrix}$$

In this example, A is a 1x3 matrix, that is, a row containing three elements. B is the transpose of A, and is therefore a 3x1 matrix, a column containing three elements.

Dimensioning the Arrays

When the TRN function is performed, the result is a new matrix, which must be assigned to an array variable, called the target variable. Both the target variable and parameter variable must be dimensioned in a DIM statement, using two subscripts to indicate the number of rows and columns. The subscripts used to dimension the target variable must be the reverse of the subscripts used to dimension the parameter variable. For instance:

```
DIM A(5,6),B(6,5)
```

This statement dimensions a 5x6 matrix A which may be used as a parameter for the TRN function. Matrix B may be used as the target variable, because it is dimensioned to be a 6x5 matrix.

A square matrix (a matrix having the same number of rows as columns) can serve as its own target variable:

```
DIM A(4,4)
A=TRN(A)
```

When the parameter variable is not square, using the same array as the target variable causes a SHAPE error.

GRAPHICS

Section 9

GRAPHICS

INTRODUCTION TO GRAPHICS

Graphic extensions to this BASIC language are very powerful and easy to use. The following is a summary of the graphic concepts discussed in this section.

User Data Units

The Graphic System allows you to draw graphs using the unit of measure appropriate to the application. The term "user data units" refers to these units of measure. For example, if you're programming the system to draw a sales graph, you might select dollars for the unit of measure on the vertical axis and months for the unit of measure on the horizontal axis. The term "user data units" in this case refers to "dollars" and "months." User data units for each application are established with the WINDOW statement.

Graphic Display Units

Internal screen units are based on the graphic display unit concept. Internally, the BASIC interpreter always converts graphic coordinates specified in user data units to graphic display units before addressing the screen. A graphic display unit is defined as 1/100 of the shortest axis on the drawing surface. Once the BASIC interpreter converts the data to graphic display units, the data is device independent. That is, a set of data which is recorded in GDUs can be plotted on any graphic surface regardless of the physical dimensions of the surface. The data is automatically scaled to fit the surface.

The Viewport

The viewport is defined as the drawing surface upon which graphic data is plotted. The VIEWPORT statement establishes the boundaries of the drawing surface on the GS display or on an external peripheral device. This surface can be any rectangular shape and can be located anywhere on the screen.

The Window

The window is the image of the viewport cast onto the user data space. The WINDOW statement specifies what portion of the data fits into the viewport and allows the coordinates of different points on the screen to be specified in user data units (inches, feet, miles, dollars, pounds, etc.).

Setting Up a Scale

The SCALE statement specifies how many user data units fit inside a graphic display unit. Although this ratio can be set with the VIEWPORT and WINDOW statements, the SCALE statement provides a quick and easy way to establish the ratio.

The Graphic Cursor

The POINTER statement causes the BASIC interpreter to display the graphic cursor (a blinking arrow) instead of a 5X8 matrix cursor. The graphic cursor marks the exact location of the graphic point, and can be moved to any location on the display by rotating the handle on an optional peripheral device called a Joystick. The graphic cursor goes away when a keyboard key is pressed and the coordinates of the graphic point are recorded.

Graphic Input

The GIN statement records the location of the lower-left corner of the alphanumeric cursor on the GS display. The X and Y coordinates of the point are assigned to two variables.

Absolute and Relative Moves

The MOVE and RMOVE statements cause the GS display cursor to move to any point within the defined viewport. The absolute coordinates of the destination point are specified as parameters in the MOVE statement; however, the RMOVE statement only requires the relative horizontal and vertical distance to the destination point. The parameters for both statements are specified in user data units.

Absolute and Relative Draws

The BASIC interpreter draws a vector (a line) on the GS display by executing a DRAW statement or a RDRAW statement. The vector is drawn from the present position of the cursor to the specified destination point. In the DRAW statement, the absolute coordinates of the destination point are specified; however, the RDRAW statement only requires the relative horizontal and vertical distance from the present position of the cursor. The parameters for both statements are specified in user data units.

Matrix Draws

"Fast Graphics" are executed by storing data points in two arrays; the X values in one array and the Y values in another. The array variables representing the two arrays are specified in a DRAW statement as X and Y parameters and the BASIC interpreter executes a series of draws, one for each pair of elements in the arrays.

Rotating Vectors

The ROTATE statement causes the BASIC interpreter to rotate a relative vector around its starting point. Any rotation angle can be specified.

A Ready Made Axis

The AXIS statement causes the BASIC interpreter to draw an X-Y axis on the GS display. The intercept point (point of origin) can be specified anywhere on the screen and the interval between tic marks on the axes can be specified to be any distance.

Printing Alphanumeric Characters

The PRINT statement causes a graphic device to print the specified alphanumeric characters.

External Graphic Devices

The statements just discussed can be directed toward an external peripheral device on the GPIB by specifying the appropriate I/O address in the graphic statement. There are two graphic functions, however, that can be directed toward an external graphic device, but not directed toward the GS display. One function is the "ALPHASCALE" parameter setting which sends alphanumeric scale information to an external peripheral device. This parameter information established the size of the alphanumeric characters which are printed on the graphic surface. The other function is the "ALPHAROTATE" parameter which sends alphanumeric rotation information to an external peripheral device. This information sets an internal parameter which causes all alphanumeric information to be printed at an angle on the graphic surface.

THE "ALPHAROTATE" PARAMETER

Purpose

The "ALPHAROTATE" parameter sends alphanumeric rotation information to an external peripheral device on the General Purpose Interface Bus.

Explanation

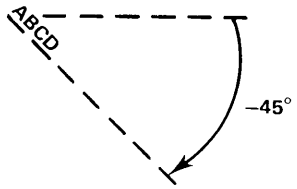
The Graphic System has the ability to send alphanumeric rotation information to an external peripheral device on the General Purpose Interface Bus (GPIB). The peripheral device receiving the information must have the facility to rotate alphanumeric characters accordingly. The GS display does not have this facility.

Alphanumeric rotation information is sent to an external peripheral device through a special PRINT statement. For example:

```
SET DEG
PRINT @16,25:-45
```

The statement SET DEG sets the trigonometric units for the system to degrees. The special PRINT statement then sends the alphanumeric rotation information to peripheral device 16 on the GPIB. The I/O address @16,25: is sent first. Primary address 16 tells device 16 to prepare to take part in an I/O operation. Secondary address 25 tells device 16 that the BASIC interpreter is about to send alphanumeric rotation information in the form of an ASCII character string.

The rotation angle is specified after the colon (:) in the special PRINT statement. The BASIC interpreter converts this information into an ASCII character string and sends the string to the specified peripheral device; in this case, device 16. It is up to device 16 to receive the ASCII string and set its internal rotation parameter to −45 degrees. The results are shown below:



Since this is an environmental command, an immediate result may not be seen while alphanumeric characters are printed by the receiving device. When the characters are printed, they are printed at a -45° angle to the horizontal as shown in the diagram. All characters printed by this device are printed at this angle until the "ALPHAROTATE" parameter is changed.

Actually, anything can be specified after the colon in the PRINT statement; it doesn't have to be the rotation angle. The key to setting the "ALPHAROTATE" parameter is the secondary address 25. This address tells the peripheral device to treat the ASCII character string as alphanumeric rotation information. The specified ASCII character string, whatever it is, must have meaning to the peripheral device and it is up to the peripheral device to set the rotation angle accordingly.

THE "ALPHASCALE" PARAMETER

Purpose

The "ALPHASCALE" parameter sends alphanumeric scale information to an external peripheral device on the General Purpose Interface Bus.

Explanation

The Graphic System has the ability to send Alphanumeric scale information to an external peripheral device on the General Purpose Interface Bus (GPIB). The peripheral device receiving the information must have the ability to interpret the information as alphanumeric scale information and set its internal scale parameters accordingly. The alphanumeric is sent via a special PRINT statement. For example:

```
PRINT @16,17:X,Y
```

When this statement is executed, the I/O address @16,17: is sent over the GPIB. Primary address 16 tells peripheral device number 16 that it has been selected to take part in an I/O operation. Secondary address 17 tells peripheral device 16 that the information it is about to receive is alphanumeric scale information. The BASIC interpreter then converts the data items which follow the colon in the PRINT statement into an ASCII character string, and sends the character string to the specified peripheral device. In this case, the numeric value assigned to the variable X is sent first, followed by the numeric value assigned to the variable Y. Device 16 receives the ASCII string and interprets the first value as the horizontal scale factor in GDUs; the second numeric value is assumed to be the vertical scale factor in GDUs.

Actually, any type of data can be specified after the colon in the PRINT statement as long as the information can be interpreted by the receiving peripheral device as alphanumeric scale information. The key item in this PRINT statement is the secondary address 17. This secondary address tells the peripheral device to treat the ASCII data as alphanumeric scale information and to set its internal scale parameters accordingly.

THE AXIS STATEMENT

Syntax Form:

[Line number] AXI [I/O address] [numeric expression , numeric expression
[, numeric expression , numeric expression]]

Descriptive Form:

[Line number] AXIS [I/O address] [X axis tic interval in user data units ,
Y axis tic interval in user data units [, X axis intercept in user data units ,
Y axis intercept in user data units]]

Purpose

The AXIS statement executes a series of moves and draws to produce an X-Y axis on the GS display. The X-Y intercept point and tic mark intervals can be specified as parameters.

Explanation

Axes Without Tic Marks

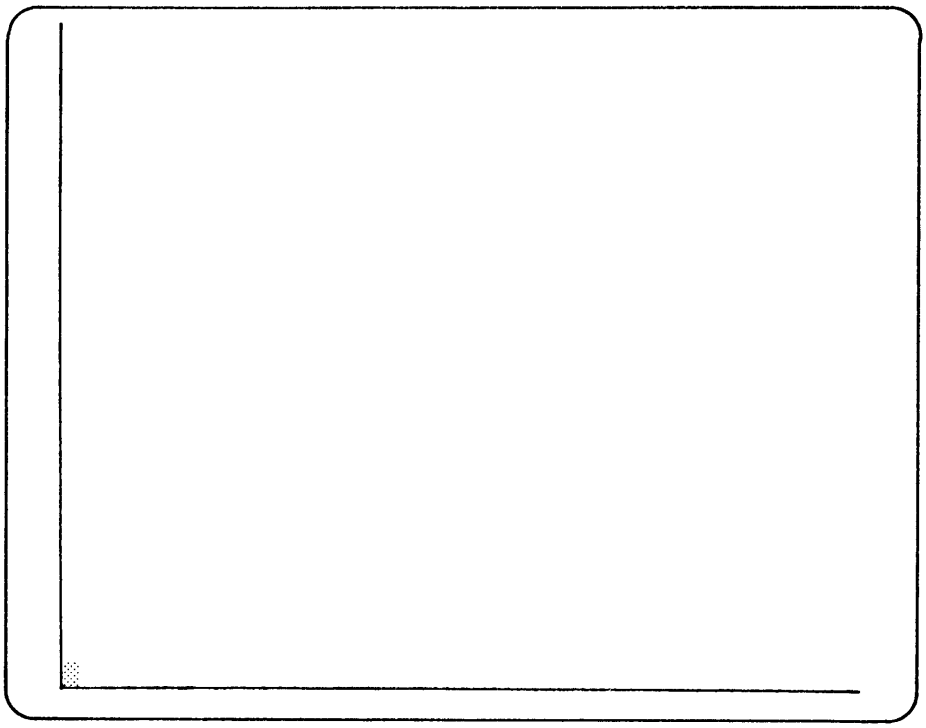
If parameters are not specified in an AXIS statement, then two lines are drawn on the GS display—one for the horizontal (X) axis and one for the vertical (Y) axis. The axes intercept each other at the point of origin (Ø,Ø). For example:

```
1ØØ INIT
11Ø AXIS
```

When line 1ØØ is executed, the WINDOW and VIEWPORT parameters are set to their default values. The viewport is set to maximum screen size and the window parameters are set to Ø,13Ø,Ø,1ØØ. This, of course, establishes a one-to-one scale factor between user data units and graphic display units.

When line 11Ø is executed, the BASIC interpreter draws two lines on the GS display; one for the horizontal axis and one for the vertical axis. The results are shown below:

GS Display Output

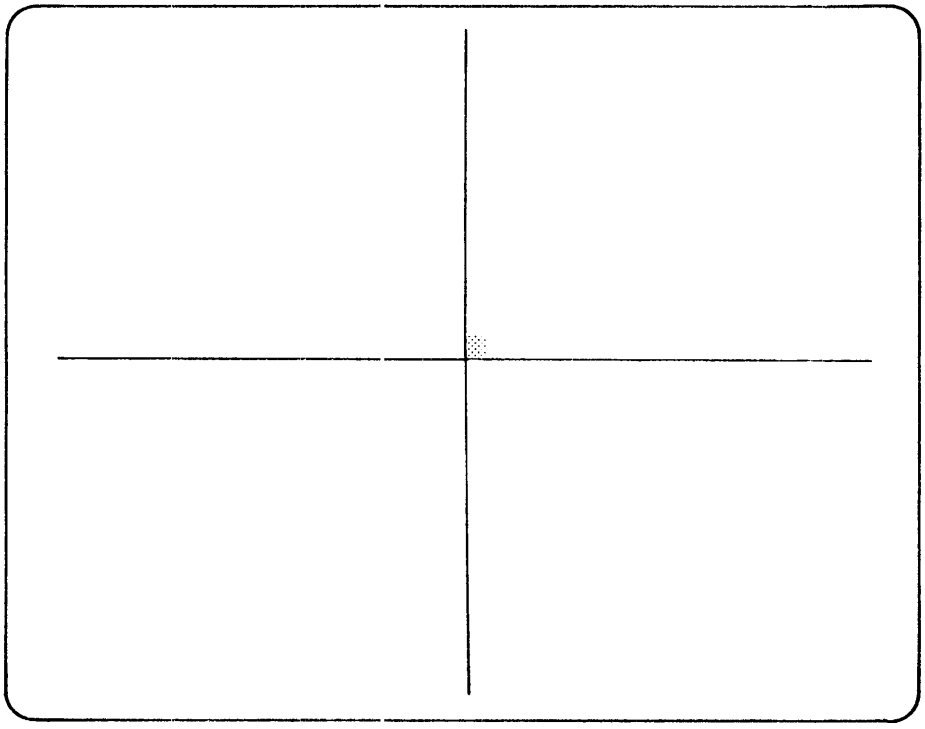


Because the point of origin is defined as the lower left corner of the window, the axes intersect each other at this point. The X axis runs along the bottom of the window; the Y axis runs along the left side of the window. Both axes are clipped at the edges of the viewport. Notice also that the alphanumeric cursor finishes at the point of origin.

Here's another example:

```
120 INIT
130 WINDOW-50,50,-100,100
140 AXIS
```

GS Display Output

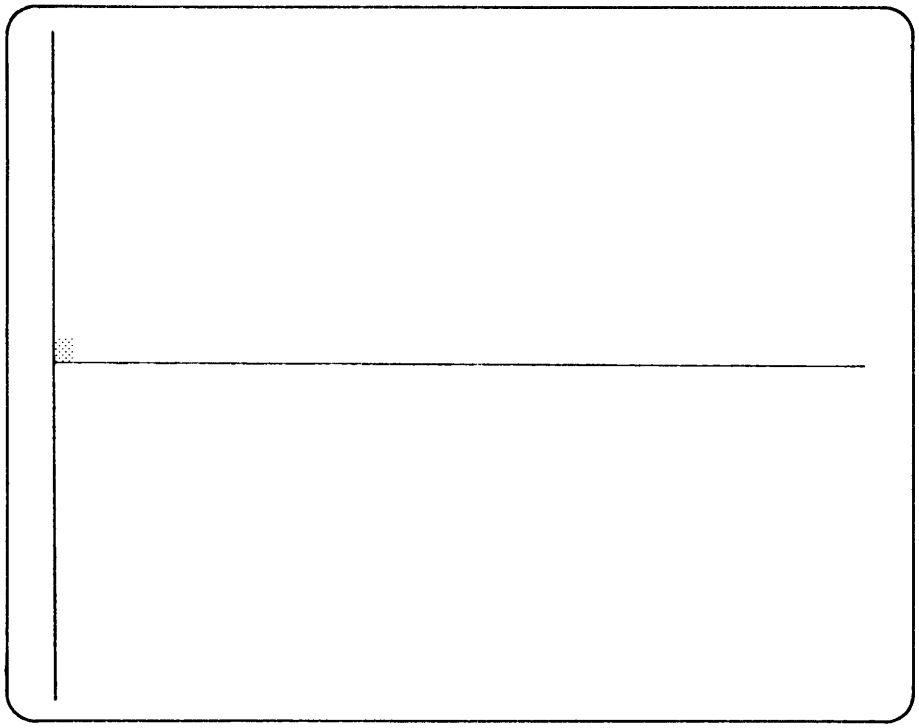


This time the WINDOW statement in line 130 defines the point of origin as the center of the window. The results are shown in the illustration. Again notice that the alphanumeric cursor finishes at the point of origin.

If the numeric range on either axis does not pass through zero, then the axis intercept point occurs at the minimum algebraic value on that axis. For example:

```
14Ø INIT
15Ø PAGE
16Ø WINDOW −1ØØ,−5Ø,−5Ø,5Ø
17Ø AXIS
```

GS Display Output



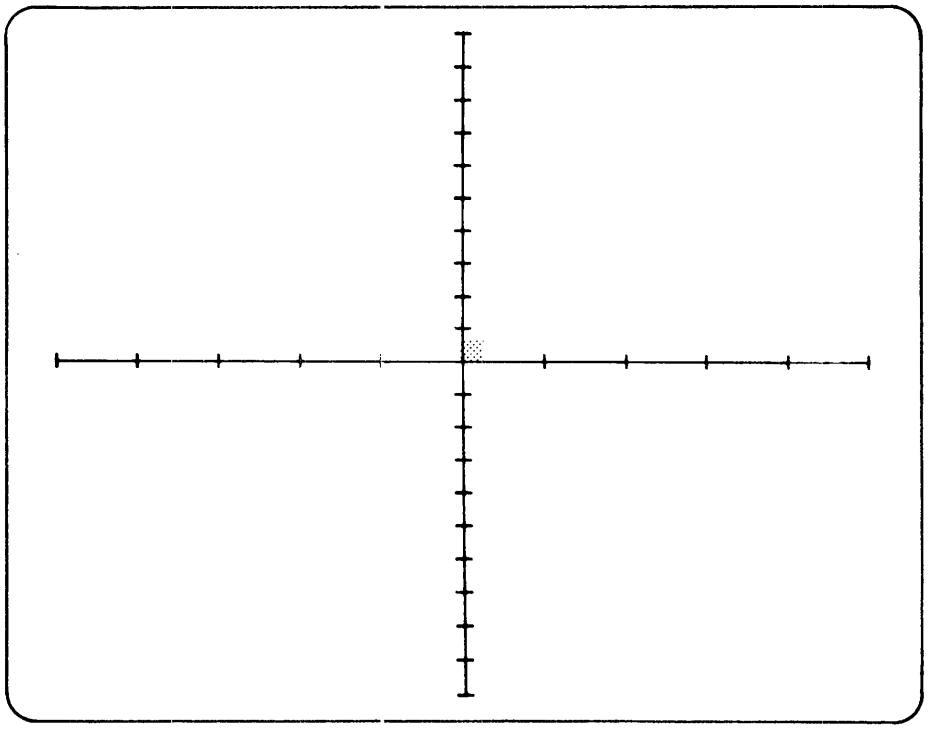
In this case, the horizontal range starts at −1ØØ and ends at −5Ø. Because this range doesn't pass through zero, the vertical (Y) axis is drawn through −1ØØ, the minimum algebraic value. The vertical range starts at −5Ø and ends at +5Ø, so the horizontal axis is drawn through the zero mark on the vertical axis.

Specifying Tic Mark Intervals

If two parameters are specified in the AXIS statement, then "tic marks" are drawn on the axis. The tic marks are used to reference the units of measure on that axis. The first parameter specifies the horizontal (X) tic interval in user data units. The second parameter specifies the vertical (Y) tic interval in user data units. For example:

```
180 INIT
190 PAGE
200 WINDOW -50,50,-100,100
210 AXIS 10,10
```

GS Display Output



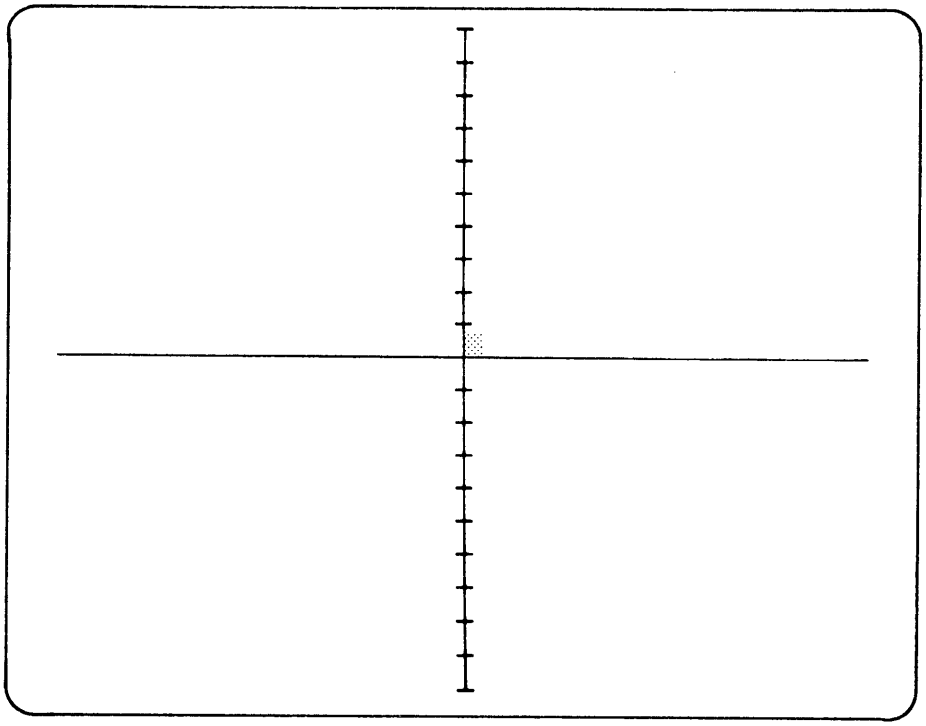
In this example, the X and Y tic mark intervals are specified as ten units each. Because the X axis numeric range extends from −50 to +50, 10 tic marks are drawn on the X axis; 5 tics on each side of the point of origin. Each tic mark represents 10 units. On the vertical axis, each tic mark also represents 10 units; however, the numeric range extends from −100 to +100, so twice as many tic marks are drawn.

Specifying a Zero Tic Mark Interval

If the tic mark interval on either axis is specified as Ø, then tic marks are not drawn on that axis. For example:

```
22Ø INIT
23Ø PAGE
24Ø WINDOW -5Ø,5Ø,-1ØØ,1ØØ
25Ø AXIS Ø,1Ø
```

GS Display Output

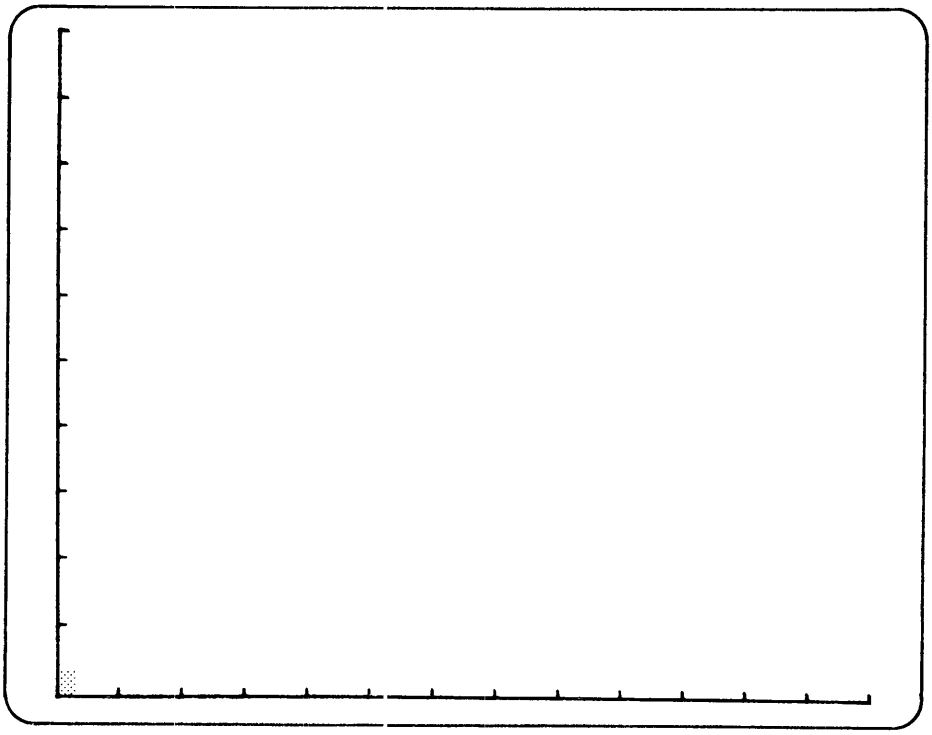


This program is identical to the previous program, except that Ø is specified as the X tic mark interval. The tic marks are suppressed on the X axis as shown in the illustration.

The length of the tic marks are scaled to 1% of the viewport size in the corresponding direction. The tic marks are normally centered on the axis unless the axis runs along the edge of the viewport; in this case, the portion of the tic mark closest to the edge of the viewport is clipped off. For example:

```
26Ø INIT
27Ø PAGE
28Ø AXIS 1Ø,1Ø
```

GS Display Output



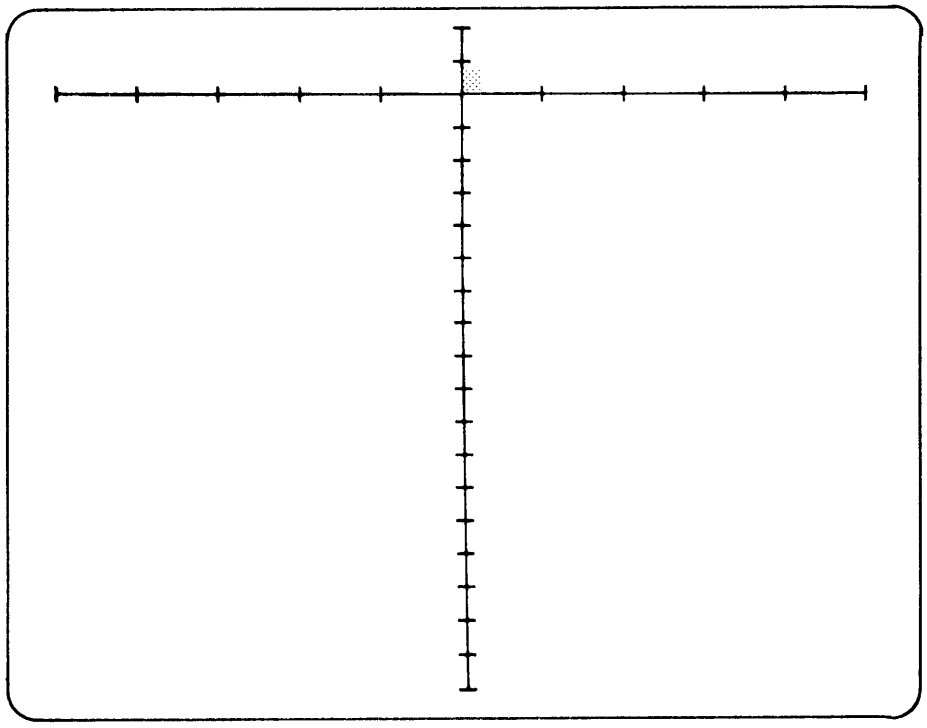
In this example, the outside portion of the tic marks are clipped off because the axes run along the edge of the viewport. The length of each tic mark is .5% of the viewport in the corresponding direction.

Specifying the Axes Intercept Point

The X-Y axis intercept point can be specified by adding two parameters after the X and Y tic mark interval specification. The third parameter specifies the X axis intercept point in user data units; the fourth parameter specifies the Y axis intercept point in user data units. For example:

```
290 INIT
300 PAGE
310 WINDOW -50,50,-100,100
320 AXIS 10,10,0,80
```

GS Display Output

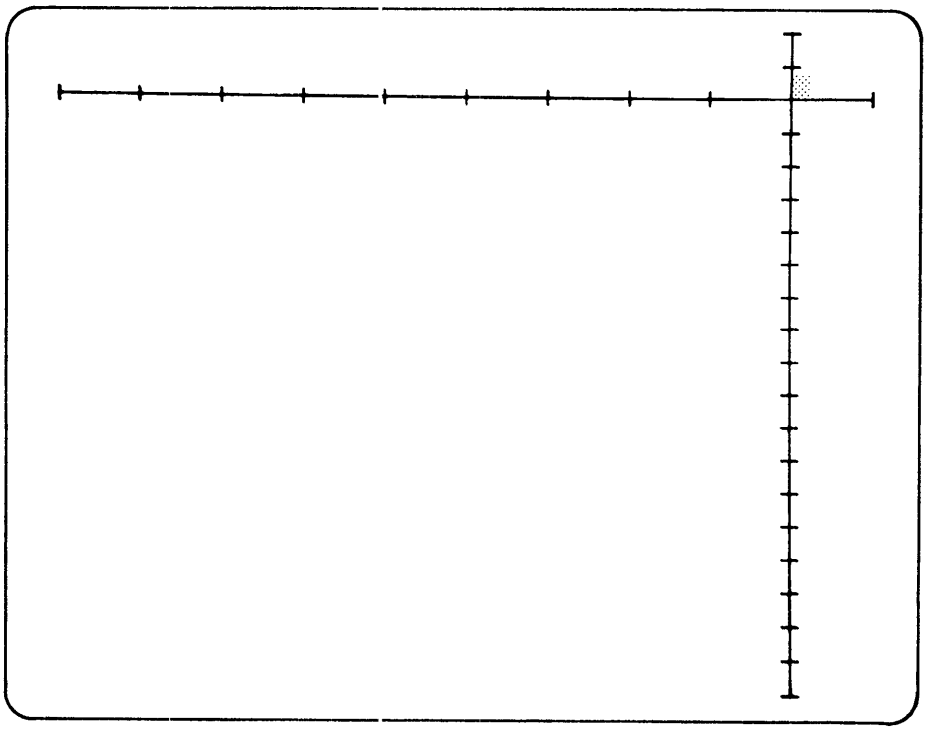


In this example, the X axis numeric range starts at −50 and ends at +50; the Y axis range starts at −100 and ends at +100. The tic mark intervals are set at 10 units on each axis. Because the third parameter in the AXIS statement is 0, the Y axis intercepts the X axis at 0. The fourth parameter is specified as 80, so the X axis crosses the Y axis at the 80 units mark.

Another example:

```
330 INIT
340 PAGE
350 WINDOW -50,50,-100,100
360 AXIS 10,10,40,80
```

GS Display Output



This program produces the same results as the last program, except that the X-Y intercept point is changed. This time the X axis intercept point is specified as 40 in line 360; the Y axis intercept is specified as 80. The results are shown in the illustration.

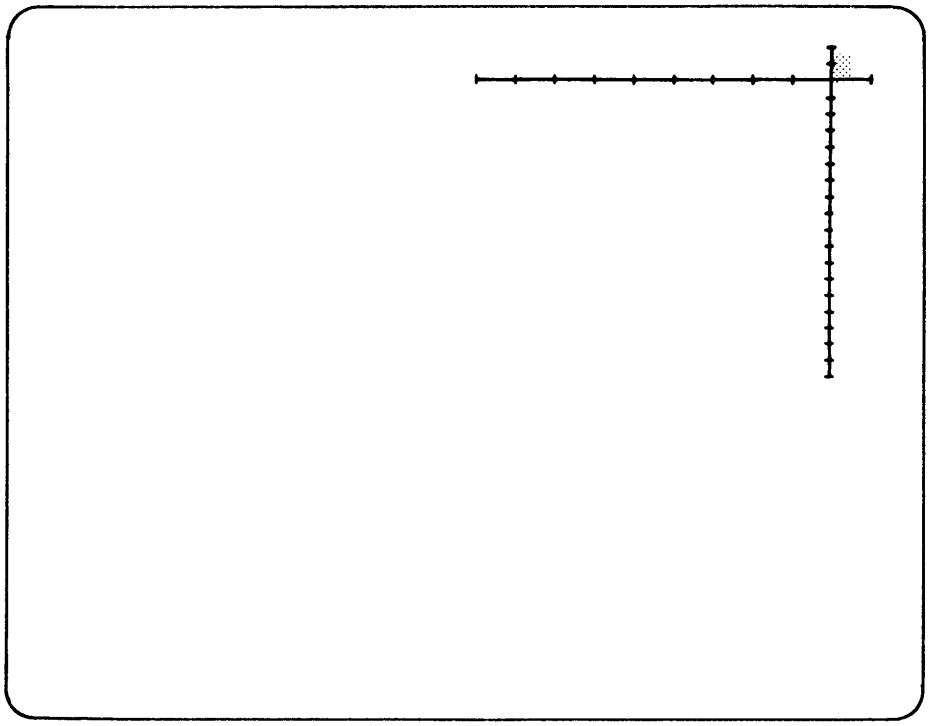
If an intercept point is specified beyond the numeric range established for the WINDOW, the axis passing through that intercept point is not drawn. In this case, if the third parameter is 60 instead of 40, the X axis isn't drawn because it lies outside the window.

Changing the Size of the Viewport

If the viewport is reduced in size, the axes are automatically scaled to fit into the new viewport. For example:

```
360 PAGE
370 VIEWPORT 65,130,50,100
380 WINDOW -50,50,-100,100
390 AXIS 10,10,40,80
```

GS Display Output



In this example, the viewport is defined as the upper-left corner of the screen. The same WINDOW and AXES are defined as the previous program. The results are shown in the illustration. Notice that the axes are automatically reduced in size to fit into the smaller viewport.

Labeling Tic Marks on an Axis

The tic marks on an axis can be labeled by executing a MOVE to a particular point above or below the tic mark, then executing a PRINT statement to print the appropriate label. For detailed information on axis labeling techniques, refer to the PLOT 50 Introduction to Graphic Programming in BASIC manual.

THE DRAW STATEMENT

Syntax Form:

[Line number] DRA [I/O address] numeric expression , numeric expression

Descriptive Form:

[Line number] DRAW [I/O address] X coordinate in user data units , Y coordinate in user data units

Purpose

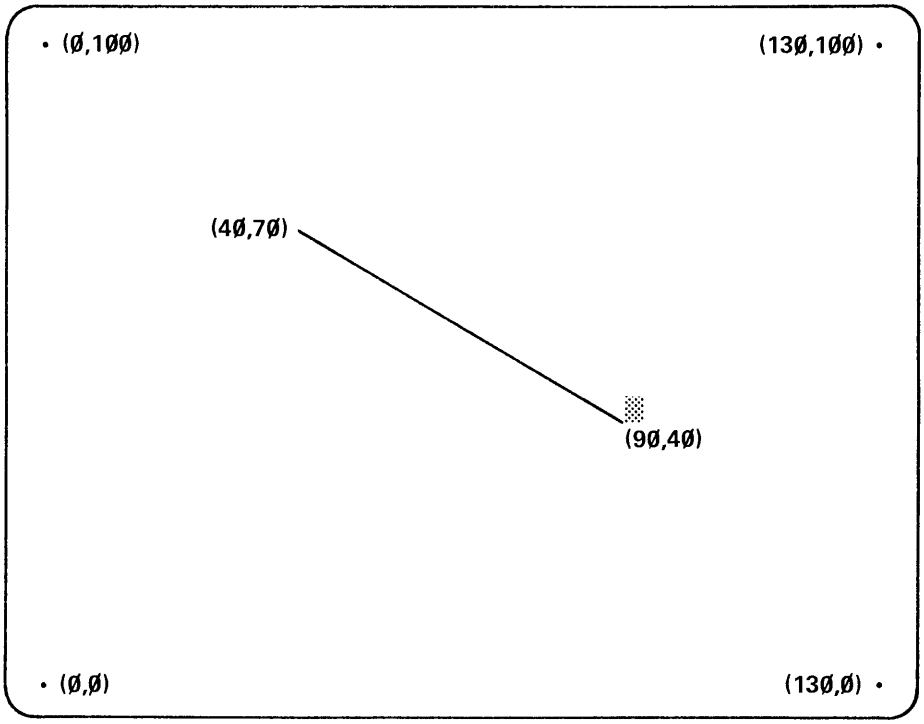
The DRAW statement reduces the specified numeric expressions to numeric constants, interprets the numeric constants as the X and Y coordinates of a graphic data point in user data units, then draws a vector (a line) on the GS display from the present position of the cursor to the specified graphic data point.

Explanation

The DRAW statement draws a vector from the present position of the cursor to the specified data point. The coordinates of the data point are specified in user data units; the lower-left corner of the alphanumeric cursor acts as the writing tool. (Refer to the User Data Unit Concept explanation in this section for a definition of user data units.) The following program illustrates the execution of a DRAW statement.

```
100 INIT
110 PAGE
120 MOVE 40,70
130 DRAW 90,40
140 END
```

GS Display Output



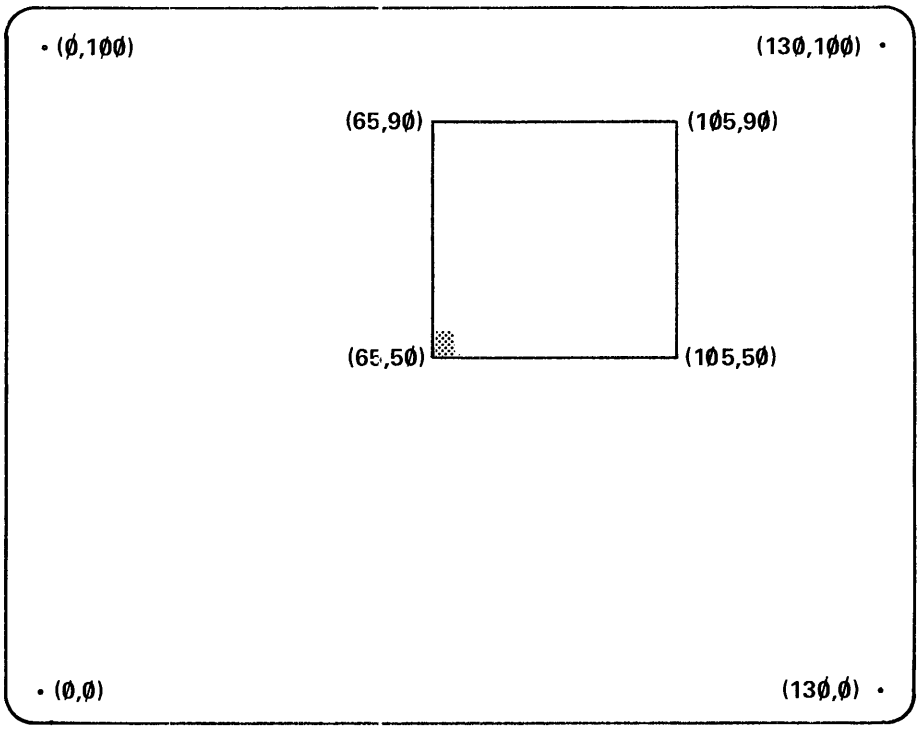
Line 1ØØ in this program sets the VIEWPORT and WINDOW parameters to their default values (for ease of illustration). Line 11Ø clears the screen and line 12Ø moves the lower-left corner of the cursor to the coordinates (4Ø,7Ø). This positions the cursor at the starting point of the vector. Line 13Ø then draws a vector from the present position of the cursor to the specified coordinates (9Ø,4Ø). The first coordinate (9Ø) specifies the absolute horizontal position of the destination point. The second coordinate (4Ø) specifies the absolute vertical position of the destination point. Both values are specified in user data units. If the point lies outside the specified WINDOW, the BASIC interpreter makes an attempt to draw to the specified coordinates; however, clipping occurs at the boundary of the VIEWPORT. (Refer to the WINDOW statement in this section for an explanation of clipping.)

Matrix DRAW

Several vectors can be drawn all at once by specifying two arrays as parameters in a DRAW statement. The following program illustrates the execution of this kind of DRAW. The DRAW statement in this program is equivalent to executing four separate DRAW statements.

```
1ØØ INIT
11Ø PAGE
12Ø DIM X(4),Y(4)
13Ø DATA 1Ø5,1Ø5,65,65,5Ø,9Ø,9Ø,5Ø
14Ø READ X,Y
15Ø MOVE 65,5Ø
16Ø DRAW X,Y
17Ø END
```

GS Display Output



In line 12Ø, the variables X and Y are dimensioned as one dimensional arrays with four elements each. The values for each element are stored in a DATA statement (line 13Ø) and assigned to each element with a READ statement (line 14Ø). The first element in array X is assigned the value 1Ø5, the second element is assigned the value 1Ø5, the third element is assigned the value 65, and so on. (Refer to the section on Input/Output Operations for a complete explanation of the READ statement and the DATA statement.)

Line 15Ø moves the cursor to the center of the viewport (65,5Ø) and the DRAW statement in line 16Ø causes the BASIC interpreter to draw the square as shown in the illustration. This DRAW statement produces the same result as four separated DRAW statements because array X and array Y contain four elements each. The first vector is drawn using X(1) as the horizontal coordinate and Y(1) as the vertical coordinate; the second vector is drawn immediately afterwards using X(2),Y(2) as coordinates, and so on. This process continues until the last element in each array is used to draw a vector or until the elements in one array run out. Both arrays should have the same number of elements. The following table summarizes the effect of the DRAW:

VECTOR	ARRAY X	ARRAY Y	DRAW EQUIVALENT
1st Vector	X(1) = 1Ø5	Y(1) = 5Ø	DRAW 1Ø5,5Ø
2nd Vector	X(2) = 1Ø5	Y(2) = 9Ø	DRAW 1Ø5,9Ø
3rd Vector	X(3) = 65	Y(3) = 9Ø	DRAW 65,9Ø
4th Vector	X(3) = 65	Y(4) = 5Ø	DRAW 65,5Ø

If a two dimension array is specified as either the X or Y coordinate, then the elements are read in row major order. The following table summarizes the results of a DRAW when X is a two dimensional array. The result is the same as the DRAW just described.

DRAW X,Y

VECTOR	ARRAY X	ARRAY Y	DRAW EQUIVALENT
1st Vector	X(1,1) = 1Ø5	Y(1) = 5Ø	DRAW 1Ø5,5Ø
2nd Vector	X(1,2) = 1Ø5	Y(2) = 9Ø	DRAW 1Ø5,9Ø
3rd Vector	X(2,1) = 65	Y(3) = 9Ø	DRAW 65,9Ø
4th Vector	X(2,2) = 65	Y(4) = 5Ø	DRAW 65,5Ø

DRAW to an External Peripheral Device

A DRAW command is sent to an external graphic device by specifying the appropriate I/O address. For example:

```
34Ø DRAW @16:3Ø,4Ø
```

In this statement, peripheral device number 16 on the General Purpose Interface Bus (GPIB) is specified as the target to receive the DRAW coordinates. When this statement is executed, the BASIC interpreter issues the primary address 16 over the GPIB. (Device 16 can be any graphic device such as an X-Y Plotter.) After the primary address is issued, the BASIC interpreter issues the secondary address (2Ø) over the GPIB. This tells the device 16 that the BASIC interpreter is executing a DRAW statement and to prepare to receive the X and Y coordinates of a graphic data point in GDUs (graphic display units). The BASIC interpreter then converts the specified X and Y coordinates from user data units to graphic display units. The X coordinate is sent over the GPIB first in ASCII code (most significant digit first); the Y coordinate follows in ASCII code (most significant digit first). After the coordinates are received by the specified device, the device draws a vector from the present position of its writing tool to the specified coordinates.

Specifying the DRAW Coordinates in GDUs

The coordinates of a DRAW statement can be specified directly in GDUs and sent to the GS display or an external peripheral device via the PRINT statement. A draw of this type is faster because the transformation from user data units to graphic display units is eliminated. The draw is executed as follows:

```
PRINT @32,2Ø: 8Ø,8Ø
```

When this statement is executed, the BASIC interpreter issues the primary address 32 and the secondary address 2Ø. This tells the GS display to execute a DRAW operation after it receives the X and Y coordinates of a graphic data point. After the address is issued, the parameters (8Ø,8Ø) are converted to an ASCII character string using the default PRINT format and are sent to the GS display. As far as the BASIC interpreter is concerned, the parameters are being sent to the GS display for printing. The GS display, however, interprets the information as the X and Y coordinates of a data point in the GDUs because it received the secondary address 2Ø. Once the information is received, the GS display draws a vector from the present position of the cursor to the coordinates (8Ø,8Ø). The WINDOW and VIEWPORT parameters have no effect on this kind of DRAW.

More than one vector can be drawn in this fashion by specifying more than one pair of coordinate values. For example:

```
PRINT @32,2Ø:8Ø,8Ø,5Ø,3Ø
```

When this statement is executed, the GS display draws two vectors. The first vector is drawn from the present position of the alphanumeric cursor to the coordinates (8Ø,8Ø). The second vector is drawn from the coordinates (8Ø,8Ø) to the coordinates (5Ø,3Ø). Remember, these values are in GDUs. The WINDOW parameters have no effect on the draw position.

If an array is specified in the PRINT statement, then the elements in the array are paired off in row major order and used as X-Y coordinates. For example:

```
PRINT @32,2Ø:A
```

When this statement is executed, array A is sent to the GS display as a sequential series of X-Y coordinates. If A has one dimension, then the elements A(1) and A(2) are used as the first X-Y coordinates; the elements A(3) and A(4) are used for the second X-Y coordinates, and so on. If array A has two dimensions, two rows and four columns for example, then the elements A(1,1) and A(1,2) are used as the coordinates for the first vector, the elements A(1,3) and A(1,4) as the coordinates for the second vector, the elements A(2,1) and A (2,2) for the third vector, and so on. Notice that this method of using arrays to draw vectors is different than the method used in the DRAW statement.

THE GIN STATEMENT

Syntax Form:

[Line number] GIN [I/O address] numeric variable , numeric variable

Descriptive Form:

[Line number] GIN [I/O address] target variable for X coordinate

, target variable for Y coordinate

Purpose

The GIN (Graphic Input) statement records the current position of the cursor on the GS display or the current position of the graphic writing tool on an external peripheral device.

Explanation

The GS Display

The GIN statement is used to find the current position of the cursor on the GS display. For example:

```
250 GIN X,Y
```

When this statement is executed, the BASIC interpreter locates the present position of the lower-left corner of the alphanumeric cursor. The X coordinate of the position is assigned to the first variable specified in the GIN statement; the variable X in this case. The Y coordinate of the position is assigned to the second variable specified in the GIN statement; the variable Y. Both values are recorded in the user data units currently specified in the WINDOW statement. The coordinate values are recorded to 3 digits of accuracy.

A Sample Program

The following program provides an illustration on how the GIN statement can be used. The program randomly moves the graphic point around the screen, prints an arrow symbol to indicate the position of the point, and prints the coordinates of the point in user data units.

Each time the RETURN key is pressed, a new point is displayed at random. If the E key is pressed, the program returns control back to the GS keyboard. Here's the program:

```
100 INIT
110 PAGE
120 PRINT "Press RETURN for a new data point"
130 PRINT "Press E to end the program"
140 PRINT @32,18:5
150 MOVE 100*RND(-2),100*RND(-2)
160 PRINT "|";
170 GIN A,B
180 PRINT A;" , ";B;
190 POINTER X,Y,Z$
200 MOVE A,B
210 IF Z$="E" THEN 230
220 GOTO 150
230 PRINT @32,18:0
240 END
```

When this program is executed, line 100 initializes the system. (Refer to the INIT statement in the Environmental Control section for details.) Line 110 pages the screen and lines 120 and 130 print the operating instructions in the upper left-hand corner of the display.

Line 140 then changes the character font to the graphic character font. (Refer to "ALPHAFONT" in the Environmental Control section for details.) This is necessary to print an arrow instead of a vertical bar in line 160.

When line 150 is executed, a move to a random point on the screen is executed. Notice that the random number function is used to generate the coordinate values for the MOVE. (Refer to the RND function in the Mathematical Operations section for details.) The random numbers (0 through 1) are multiplied by 100 to give coordinate values ranging from 0 through 100. The "arrow" is printed on the display in line 160. This indicates the position of the graphic point. Because the exact location of the graphic point is unknown at this point in the program, a GIN statement is executed next.

In line 170, the X coordinate of the graphic point is assigned to the variable A and the Y coordinate is assigned to the variable B. These coordinates are printed beside the graphic point in line 180.

When line 190 is executed, program execution stops and the BASIC interpreter waits for an entry from the keyboard. The POINTER statement is used here for two reasons. First, the program stops and gives the operator a chance to study the X and Y coordinate values; second, the key entry can be made without printing the entry on the GS display.

When a key is pressed on the GS keyboard (any key), program execution continues. Line 2ØØ moves the graphic point back to its original position. This is necessary because the PRINT statement in line 18Ø moves the cursor away from the point when the coordinates are printed. (This operation could not be done if the GIN statement in line 17Ø were not executed to record the X and Y coordinates. It is included here to illustrate how to move back to a specific point after printing a message.)

When line 2ØØ moves the cursor back to its original position, a check is made in line 21Ø to see which key was pressed. If the "E" key was pressed, then control is transferred to line 23Ø where the "ALPHAFONT" environmental parameter is reset to the U.S. Font and program execution is ended (line 24Ø). If another key was pressed, then control is transferred to line 22Ø, then to line 15Ø, and the program is repeated.

Try running the program now, just for fun, and see what happens!

External Peripheral Devices

The GIN statement can be directed toward an external peripheral device on the General Purpose Interface Bus by specifying the appropriate I/O address. For example:

```
34Ø GIN @7: X,Y
```

When this statement is executed, the I/O address @7,24: is issued over the General Purpose Interface Bus (GPIB). Primary address 7 tells peripheral device number 7 that it has been selected to take part in the upcoming I/O operation. Secondary address 24 is issued by default and tells device 7 to send the X and Y coordinates of the present position of its graphic writing tool. The peripheral device responds by sending the X coordinate first, most significant digit first, as an ASCII character string over the GPIB. The BASIC interpreter assigns this value to the variable X (the first variable specified in the GIN statement). The peripheral device then sends the Y coordinate value over the GPIB as an ASCII character string (most significant digit first). The BASIC interpreter assigns this value to the variable Y (the second variable specified). The operation is terminated when the peripheral device activates the EOI (End or Identify) signal line on the GPIB or issues a Carriage Return character, or both.

NOTE

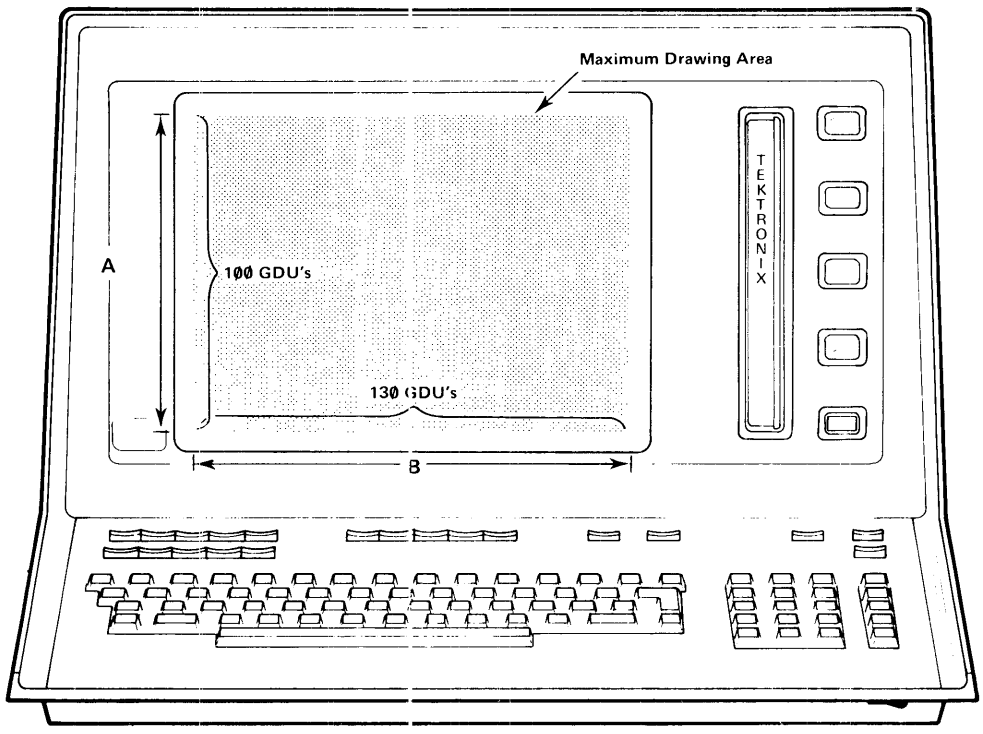
The peripheral device sends the position coordinates in graphic display units and it is up to the BASIC interpreter to convert them to user data units.

THE GRAPHIC DISPLAY UNIT CONCEPT

A graphic display unit (GDU) is an internal unit defined as one one-hundredth of the shortest axis on the drawing surface. On the GS display, the shortest axis is the vertical height of the screen. Therefore, the vertical height measures 100 GDUs. The horizontal width of the screen measures 130 GDUs. Internally, the BASIC interpreter converts all coordinate values specified in user data units to graphic display units before addressing points on the screen or the graphic surface of an external peripheral device.

The physical length of a GDU is device dependent. This means that the GDUs on one graphic device may be physically longer or shorter than the GDUs on another graphic device.

The graphic display unit concept frees graphic data from being device dependent. It provides automatic scaling which allows the same set of data points to be plotted on any graphic device without changing the numeric values to conform to the physical dimensions of the plotting surface. The example below illustrates the GDU concept.



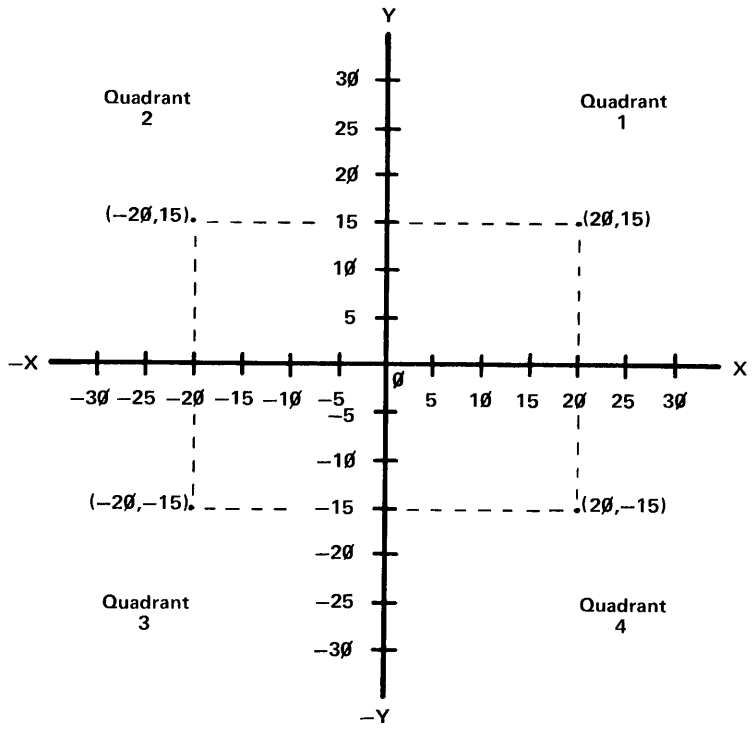
	11" display	19" display
A	7.48"	14.9"
B	5.625"	10.5"

A Word About Resolution

Since the coordinates of a data point can be specified using any decimal value, the GDU concept allows for practically infinite screen resolution. For example, the data point with the coordinates (2.ØØØ3675,4.63ØØ87) can be addressed, if so desired. In practice, however, the hardware limitations of a graphic device determine the actual resolution. On the GS display, two points must be approximately .128 GDUs apart to detect any visual separation. This resolution is equal to the resolution of Tektronix 4000-Series terminals which use the 1Ø24 X 78Ø TEK POINT concept to address points on the screen.

The Cartesian Coordinate System

In order to understand computer graphics, an understanding of the Cartesian Coordinate System is essential, so let's review it. The Cartesian Coordinate System provides a way to locate any point on a two dimensional plane. The plane is divided into four quadrants by a horizontal and vertical axis. Refer to the illustration below.



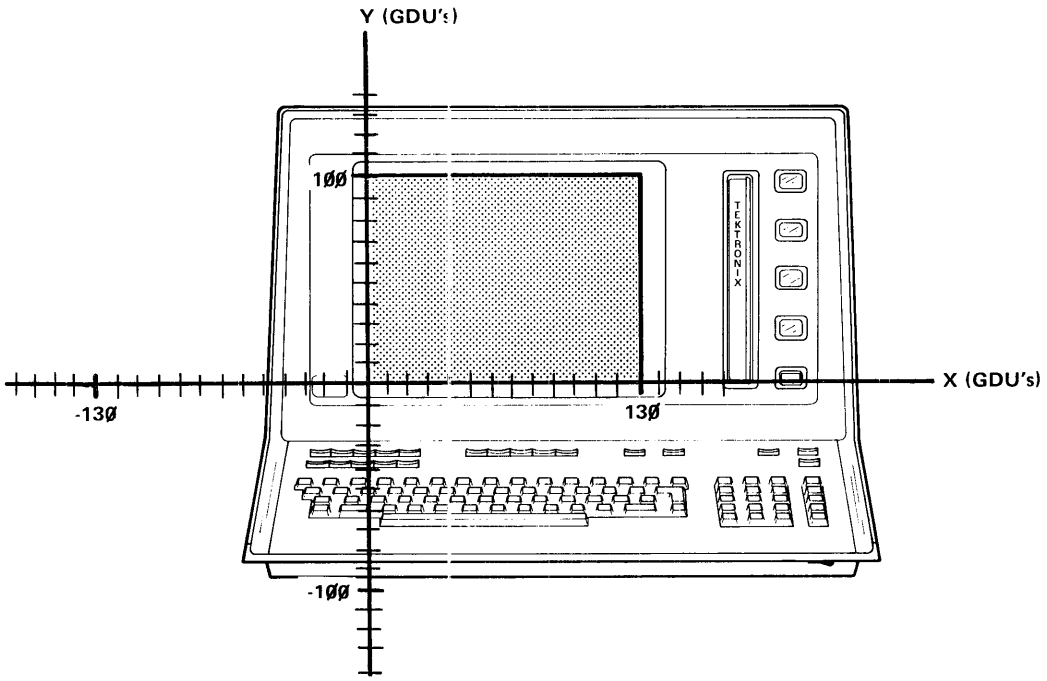
The horizontal axis is called the X axis and the vertical axis is called the Y axis. The point at which the axes cross is called the point of origin. Tic marks are used to mark the distance along each axis. Any interval between tic marks can be specified and any unit of measure can be specified. The position of each point on the plane is located by specifying two coordinates (X,Y). X represents the point's horizontal distance from the point of origin and Y represents the point's vertical distance from the point of origin.

Points located in the first quadrant are specified with a positive X value and a positive Y value (+X,+Y); the point (2Ø,15) is shown as an example. Points located in the second quadrant are specified with a negative X value and a positive Y value (−X,+Y); the point (−2Ø,15) is shown as an example. Points located in the third quadrant are specified with a negative X value and a negative Y value (−X,−Y); the point (−2Ø,−15) is shown as an example. And points located in the fourth quadrant are specified with a positive X value and a negative Y value (+X,−Y); the point (−2Ø,−15) is shown as an example.

All graphic surfaces represent a cartesian coordinate plane and the location of a point on that plane is specified by specifying the appropriate X coordinate value and the appropriate Y coordinate value.

How the GS Display Fits into a Cartesian Coordinate System

Internally, the drawing surface on the GS display represents the first quadrant in a Cartesian Coordinate System. A graphic illustration of this is shown below.



The lower-left corner of the display represents the point of origin. The vertical axis (Y) runs along the left side of the display and is marked off in Graphic Display Units. The horizontal axis runs along the bottom of the display and is also marked off in Graphic Display Units. The physical size of the drawing surface covers an area 1ØØ GDUs high by 13Ø GDUs wide. Points on the drawing surface are specified by specifying their appropriate X and Y coordinates. For example, the point (65,5Ø) is the center of the screen; the point (Ø,1ØØ) is the upper-left corner; the point (13Ø,1ØØ) is the upper-right corner; the point (13Ø,Ø) is the lower-right corner; and the point (Ø,Ø) is the lower-left corner (the point of origin). Specifying the drawing surface boundaries on the GS display with the VIEWPORT statement is based on this concept.

INPUTTING THE GRAPHIC PAGE SIZE

Syntax Form:

[Line number] INP [I/O address] { array variable / string variable / numeric variable } [, { array variable / string variable / numeric variable }] ...

Descriptive Form:

[Line number] INPUT [I/O address] target variables for incoming data items which are formatted in ASCII code

Purpose

The INPUT statement is used to record the size of the drawing surface on the GS display or the specified external peripheral device.

Explanation

The GS Display

The INPUT statement can be used to record the size of the drawing surface on the GS display. For example:

```
150 INPUT @32:X,Y
```

When this statement is executed, primary address 32 selects the GS display as the input source. The GS display can only respond with one thing—its page size. The horizontal width of the screen (130 GDUs) is sent first and assigned the numeric variable X. The vertical height of the screen (100 GDUs) is sent next and assigned to the numeric variable Y. Any two numeric variables can be specified as target variables here.

External Peripheral Devices

If a graphic device on the General Purpose Interface Bus (GPIB) is specified as the input source in an INPUT statement, the graphic device must have the capability to send the dimensions of its plotting surface to the BASIC interpreter in Graphic Display Units. The dimensions are sent over the GPIB as two ASCII character strings. The first character string represents the horizontal dimension of the plotting surface. This string is sent most significant

digit first and is assigned to the first variable specified as a parameter. The second ASCII character string represents the vertical dimension of the plotting surface. This string is also sent most significant digit first and is assigned to the second variable which is specified as a parameter. Both character strings are terminated with Carriage Return (CR). For example:

```
535 INPUT @16:A,B
```

In this example, device number 16 is selected as the input source. (Assume that device number 16 is an X-Y Plotter on the General Purpose Interface Bus.) When this statement is executed, the I/O address @16,13: is sent to the plotter over the GPIB. Secondary address 13 is sent to the plotter by default. This tells the plotter to send the dimensions of its plotting surface. The plotter responds by sending the horizontal dimension first as an ASCII character string. This value is assigned to the variable A. Next, the plotter sends the vertical dimension. This value is assigned to the variable B. Once the transfer is complete, the BASIC interpreter is free to use this information as specified in the program.

THE MOVE STATEMENT

Syntax Form:

[Line number] MOV [I/O address] numeric expression , numeric expression

Descriptive Form:

[Line number] MOVE [I/O address] X coordinate in user data units , Y coordinate in user data units

Purpose

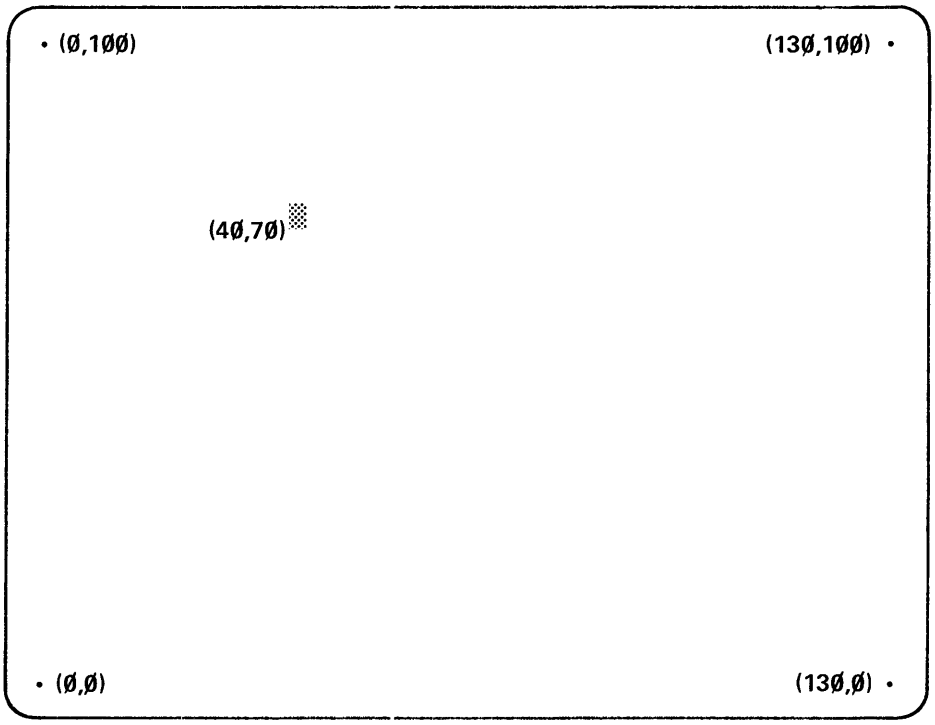
The MOVE statement reduces the specified numeric expressions to numeric constants, interprets the numeric constants as the coordinates of a graphic data point in user data units, then moves the lower-left corner of the alphanumeric cursor to the specified graphic data point.

Explanation

The MOVE statement can move the lower-left corner of the cursor to any point in the specified viewport. The following program illustrates the execution of a MOVE statement.

```
100 INIT
110 MOVE 40,70
120 END
```

GS Display Output



Line 1ØØ in this program sets the VIEWPORT and WINDOW parameters to their default values (for ease of illustration). Line 11Ø then moves the lower-left corner of the alphanumeric cursor to the coordinates (4Ø,7Ø); 4Ø is the horizontal coordinate (X) specified in user data units; 7Ø is the vertical coordinate (Y) specified in user data units. (Refer to the User Data Units Concept explanation in this section for a definition of user data units.) Moves are normally used to position the cursor at the starting point of a vector or text output and are usually executed just prior to a DRAW statement or a RDRAW statement or a PRINT statement.

Matrix MOVE

If two arrays are specified as parameters in a MOVE statement, then the BASIC interpreter executes a series of MOVEs—one move for each pair of elements in the arrays. For example, if the statement MOVE X,Y is executed and X and Y are one dimensional arrays, then the first MOVE is executed using the element X(1) as the horizontal coordinate and the element Y(1) as the vertical coordinate. The next MOVE is executed with X(2) as the horizontal coordinate and Y(2) as the vertical coordinate, and so on. A MOVE is executed for each pair of elements in the matrixes. A MOVE of this nature is of little practical value and is seldom used.

Specifying an External Peripheral Device in a MOVE Statement

A MOVE command can be sent to an external graphic peripheral device by specifying the appropriate I/O address. For example:

```
26Ø MOVE @16:5Ø,5Ø
```

In this statement device number 16 on the General Purpose Interface Bus (GPIB) is specified to receive the MOVE information. When this statement is exectued, the BASIC interpreter issues the primary address 16 over the GPIB. Device 16 can be any graphic device such as an X-Y Plotter. (The primary address assignments are pre-defined through hardware connections on the peripheral device.) After the primary address is issued, the BASIC interpreter sends the secondary address (21) over the GPIB. This tells the receiving device that the BASIC interpreter is executing a MOVE statement and to prepare to receive the X and Y coordinates of a graphic data point in GDUs (Graphic Display Units). The BASIC interpreter then converts the specified X and Y coordinates from user data units to graphic display units. The X coordinate is sent over the GPIB in ASCII code first (most significant digit first); the Y coordinate follows in ASCII code (most significant digit first). After the coordinates are received by the specified device, the device moves its writing tool to the position specified by the coordinates.

Specifying the MOVE Coordinates in GDUs

The coordinates of a MOVE statement can be specified directly in GDUs and sent to the GS display or to an external peripheral device via the PRINT statement. A move of this type is faster because the transformation from user data units to graphic display units is eliminated. A MOVE to the GS display is executed as follows:

```
PRINT @32,21:8Ø,8Ø
```

When this statement is executed, the BASIC interpreter issues the primary address 32 and the secondary address 21. This tells the GS display to execute a MOVE operation after it receives the X and Y coordinates of a graphic data point. After the address is issued, the paramters (8Ø,8Ø) are converted to an ASCII character string, using the default PRINT format, and sent to the display. As far as the BASIC interpreter is concerned, the parameters are being sent to the GS display for printing. The display, however, interprets the information as the X and Y coordinates of a data point in GDUs, because it received the secondary address 21. Once the information is recieved, the GS display moves the cursor to the coordinates (8Ø,8Ø). The WINDOW and VIEWPORT parameters have no effect on this kind of MOVE.

THE POINTER STATEMENT

Syntax Form:

[Line number] POI numeric variable , numeric variable , string variable

Descriptive Form:

[Line number] POINTER target variable for X coordinate of graphic point in user data
units , target variable for Y coordinate of graphic point in user data
units , target variable to record the key which is pressed to end the entry

Purpose

The POINTER statement places the graphic cursor (blinking arrow) on the GS display screen. The graphic cursor points to the present position of the graphic point and is moved around the screen by rotating a Joystick (an optional peripheral device). When a keyboard key is pressed, a GIN (Graphic Input) operation is executed to mark the location of the graphic point and program execution continues to the next statement.

NOTE

For the 4054 Graphic System, the graphic cursor consists of two crosshairs instead of a blinking arrow. The thumbwheels at the right side of the keyboard control the movement of the crosshairs; no Joystick is necessary.

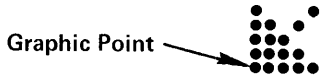
Explanation

Displaying the Graphic Cursor

The POINTER statement places the graphic cursor on the GS display screen. For example:

```
100 POINTER A,B,C$
```

When this statement is executed, the graphic cursor is placed on the GS display. The exact position of the cursor depends on the position of the Joystick . The tip of the arrow on the graphic cursor marks the current location of the graphic point. (This point is normally used as a reference to draw graphic vectors). The graphic cursor and the location of the graphic point are shown:



Rotating the Joystick moves the graphic cursor to a specific location on the screen. Rotation to the right, for example, moves the cursor to the right side of the screen.

Completing the POINTER Operation

After the graphic cursor is positioned on the screen with the Joystick, a key is pressed on the GS keyboard. When a key is pressed, the X and Y coordinates of the current location of the graphic point are recorded. The X coordinate is assigned to the first target variable specified in the POINTER statement; in this case, the numeric variable A. The Y coordinate is assigned to the second target variable specified in the POINTER statement; in this case, the numeric variable B. Both the X and Y coordinates are measured in the user data units as specified in the WINDOW statement. The third target variable records which key is pressed to terminate the operation. The key symbol is recorded, but not echoed on the GS display. In this case, if the F key is pressed after the graphic cursor is placed on the screen and the letter "F" is assigned to C$. After the key is pressed, program execution continues to the next statement.

Applications for the POINTER Statement

The POINTER statement and the Joystick provide the keyboard operator with a method to graphically communicate with the system. For example, assume that 1ØØØ data values are input into memory from an external peripheral device over the General Purpose Interface Bus. Assume also that this data is processed by the BASIC program and displayed on the GS display as a point plot graph. The POINTER statement and the Joystick allow the keyboard operator to "talk" to the system about the graph. In this case, program subroutines can be placed in memory so that the keyboard operator can place the graphic cursor over a particular data point on the graph and press a key to execute a predefined function. Pressing the "Y" key might mean "display the numeric value of this data point"; pressing the "D" key might mean "delete this data point from memory."

The POINTER statement only records the X and Y coordinates of the present position of the graphic point. It is up to the programmer to generate the appropriate response by placing program subroutines in memory.

NOTE

If the Joystick is not attached to the rear-panel connector, the pointer is positioned at random. The pointer position is independent of the cursor position.

THE PRINT STATEMENT

Syntax Form:

[Line number] PRI [I/O address] [USI { string constant / string variable / line number } :]

[{ string constant / string variable / numeric expression } [{ , / ; } string constant / string variable / numeric expression]] ... [;]

Descriptive Form:

[Line number] PRINT [I/O address] [USING { format string / format string variable / IMAGE line number } :]

[item to be printed [{ , / ; } item to be printed]] ... [;]

PURPOSE

The PRINT statement sends alphanumeric information to a graphic peripheral device.

EXPLANATION

Assume that device number 16 on the General Purpose Interface Bus is an X-Y Plotter. The following statement is used to send alphanumeric information to the plotter:

```
47Ø PRINT @16: "3.65 MILLIVOLTS"
```

This statement causes the BASIC interpreter to send the character string "3.65 MILLIVOLTS" to the plotter in ASCII code, starting with the left-most character (3) and ending with the right-most character (S). The ASCII string is terminated with a Carriage Return. The BASIC interpreter assumes the peripheral has the capability to receive and print alphanumeric information.

Once the alphanumeric information is transmitted to the plotter, the plotter prints the information on the plotting surface starting at the present position of the writing tool. If the plotter has the capability to rotate alphanumeric information, the rotation angle can be transmitted to the plotter beforehand. (Refer to the Environmental Control section or the "ALPHAROTATE" parameter in this section for a complete explanation of "ALPHAROTATE.")

THE RDRAW STATEMENT

Syntax Form:

[Line number] RDR [I/O address] numeric expression , numeric expression

Descriptive Form:

[Line number] RDRAW [I/O address] X increment in user data units , Y increment in user data units

Purpose

The RDRAW statement reduces the specified numeric expressions to numeric constants, interprets the numeric constants as the increments of a graphic data point relative to the present position of the alphanumeric cursor, then draws a vector from the present position of the alphanumeric cursor to the specified data point.

Explanation

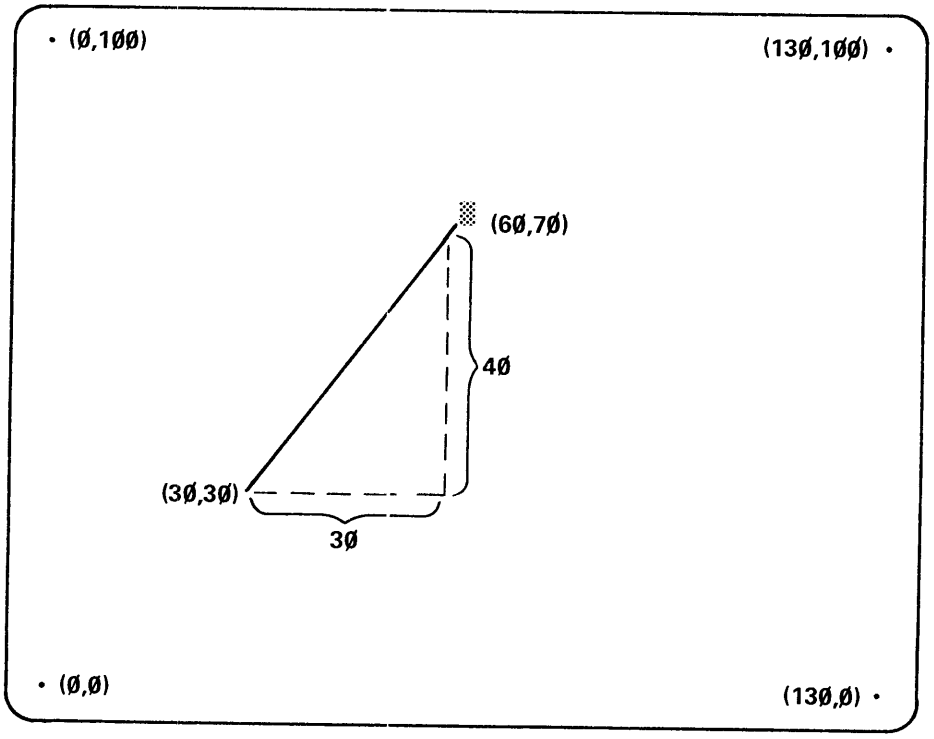
The RDRAW statement frees the programmer from having to figure out the absolute coordinates of each data point. If the next vector to be drawn is a point 3Ø units to the right and 4Ø units above the present position of the cursor, for example, then these increments are specified in a RDRAW statement. The BASIC interpreter figures out the absolute coordinates of the destination point and draws the vector to that point.

The first parameter in the RDRAW statement specifies the horizontal increment in user data units. Positive values cause movement to the right; negative values cause movement to the left. The second parameter specifies the vertical increment in user data units. Positive values cause upward movement; negative values cause downward movement. All movement is relative to the present position of the cursor and clipping occurs if the boundary of the viewport is crossed.

The following example illustrates the execution of an RDRAW statement:

```
100 INIT
110 PAGE
120 MOVE 30,30
130 RDRAW 30,40
140 END
```

GS Display Output



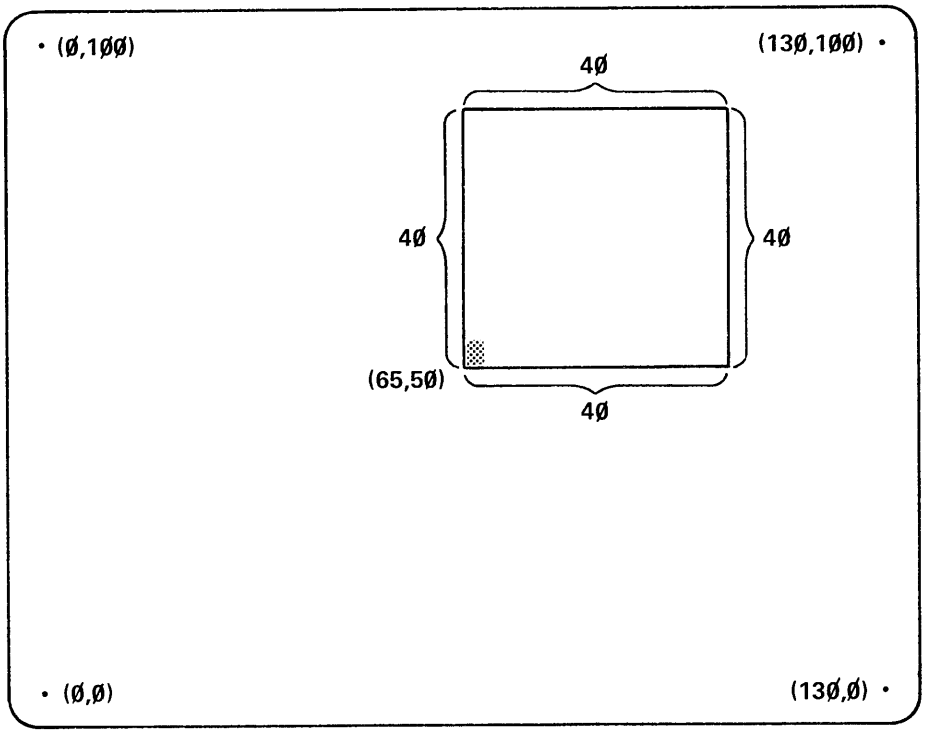
Line 1ØØ in this program sets the VIEWPORT and WINDOW parameters to their default values (for ease of illustration). Line 11Ø clears the screen. Line 12Ø moves the cursor to the absolute coordinates (3Ø,3Ø). Line 13Ø then causes the BASIC interpreter to draw a vector from the present position of the cursor (3Ø,3Ø) to a position 3Ø units to the right and 4Ø units above. The result is a vector drawn to the absolute coordinates (6Ø,7Ø). The same result can be achieved by executing the statement DRAW 6Ø,7Ø.

Matrix RDRAW

Several relative draws, like several absolute draws, can be executed in one statement by specifying arrays for coordinates. For example:

```
100 INIT
110 PAGE
120 DIM X(4),Y(4)
130 DATA 40,0,-40,0,0,40,0,-40
140 READ X,Y
150 MOVE 65,50
160 RDRAW X,Y
170 END
```

GS Display Output



In this program, the variables X and Y are defined as one dimensional arrays with four elements each (line 120). The values for the elements are stored in a DATA statement in line 130 and assigned to each element with the READ statement in line 140. The first element in array X is

assigned the value 4Ø, the second element is assigned the value Ø, and so on. (Refer to the Input/Output Operations section for a complete explanation of the READ statement and the DATA statement.)

Line 15Ø moves the cursor to the center of the viewport (65,5Ø) and the RDRAW statement in line 16Ø causes the BASIC interpreter to draw the square as shown in the illustration above. This RDRAW statement produces the same result as four RDRAW statements, because array X and array Y contain four elements each. The first vector is drawn using X(1) as the horizontal increment and Y(1) as the vertical increment; the second vector is drawn immediately afterwards using X(2) as the horizontal increment and Y(2) as the vertical increment, and so on. This process continues until the last element in each array is used to draw a vector or until one of the arrays run out of elements. Both arrays should have the same number of elements. The following table summarizes the effect of the Matrix DRAW:

RDRAW X,Y

VECTOR	ARRAY X	ARRAY Y	RDRAW EQUIVALENT
1st Vector	X(1) = 4Ø	Y(1) = Ø	RDRAW 4Ø,Ø
2nd Vector	X(2) = Ø	Y(2) = 4Ø	RDRAW Ø,4Ø
3rd Vector	X(3) = −4Ø	Y(3) = Ø	RDRAW −4Ø,Ø
4th Vector	X(4) = Ø	Y(4) = −4Ø	RDRAW Ø,−4Ø

If a two dimensional array is specified, then the elements are read in row major order. The following table summarizes the results of an RDRAW when X and Y are two dimensional arrays. The result is the same as the RDRAW just described.

RDRAW X,Y

VECTOR	MATRIX X	MATRIX Y	RDRAW EQUIVALENT
1st Vector	X(1,1) = 4Ø	Y(1,1) = Ø	RDRAW 4Ø,Ø
2nd Vector	X(1,2) = Ø	Y(1,2) = 4Ø	RDRAW Ø,4Ø
3rd Vector	X(2,1) = −4Ø	Y(2,1) = Ø	RDRAW −4Ø,Ø
4th Vector	X(2,2) = Ø	Y(2,2) = −4Ø	RDRAW Ø,−4Ø

RDRAW to an External Peripheral Device

An external peripheral device can be specified as the destination point for an RDRAW operation by specifying the appropriate I/O address in the statement. For example:

```
24Ø RDRAW @16:64,78
```

When this statement is executed, the BASIC interpreter issues the I/O address @16,2Ø: over the General Purpose Interface Bus (GPIB). Primary address (16) specifies the graphic device (an X-Y plotter for example). This primary address is preassigned to the graphic device through hardware connections. The secondary address (2Ø) is issued to the graphic device by default and tells the device to prepare to receive the X and Y coordinates from a DRAW statement.

After the I/O address is issued, the BASIC interpreter converts the increments (64,78) to the coordinates of the appropriate graphic data point. These coordinates are then converted from user data units to graphic display units (GDUs) and sent to the specified graphic device over the GPIB. The X coordinate is sent first (most significant digit first) followed by the Y coordinate (most significant digit first). Once these values are received by the graphic device, the device draws a vector from the present position of its writing tool to the specified graphic data point.

THE RMOVE STATEMENT

Syntax Form:

[Line number] RMO [I/O address] numeric expression , numeric expression

Descriptive Form:

[Line number] RMOVE [I/O address] X increment in user data units , Y increment in user data units

Purpose

The RMOVE statement reduces the specified numeric expressions to numeric constants, interprets the numeric constants as the increments of a graphic data point relative to the present position of the alphanumeric cursor, then moves the lower-left corner of the cursor to the specified data point.

Explanation

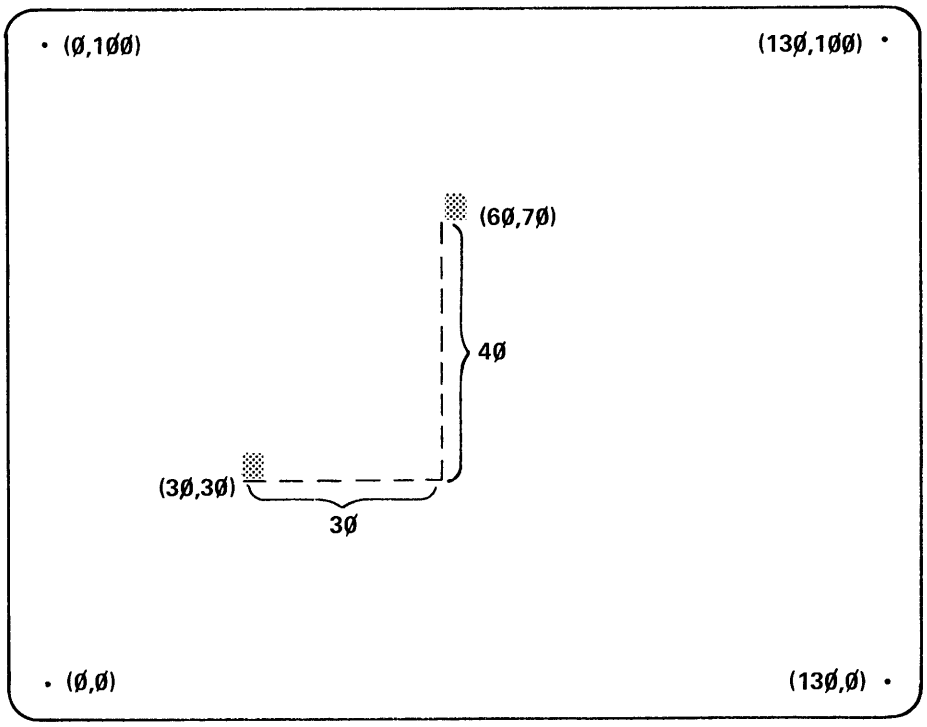
The RMOVE statement frees the programmer from having to figure out the absolute coordinates of each data point. If the cursor is to be moved to a point 3Ø units to the right and 4Ø units above its present position, for example, then these increments are specified in a RMOVE statement. The BASIC interpreter figures out the absolute coordinates of the destination point and moves the cursor to that point.

The first parameter in the RMOVE statement specifies the horizontal increment (X) in user data units. Positive values cause movement to the right; negative values cause movement to the left. The second parameter specifies the vertical increment (Y) in user data units. Positive values cause upward movement; negative values cause downward movement. All movement is relative to the present position of the cursor.

The following example illustrates the execution of a RMOVE statement:

```
100 INIT
110 PAGE
120 MOVE 30,30
130 RMOVE 30,40
140 END
```

GS Display Output



Line 100 in this program sets the VIEWPORT and WINDOW parameters to their default values (for ease of illustration). Line 110 clears the screen. Line 120 then moves the cursor to a position 30 units to the right and 40 units above the present position of the cursor. The result is a move to the absolute coordinates (60,70). The same result can be achieved by executing the statement MOVE 60,70.

Matrix RMOVE

Several relative moves, like several absolute moves, can be executed all in one statement by specifying arrays for parameters. This has little practical value, however, and is seldom used.

RMOVE to an External Peripheral Device

An external peripheral device can be specified for a RMOVE operation by specifying the appropriate I/O address. For example:

```
240 RMOVE @16:64,78
```

When this statement is executed, the BASIC interpreter issues the I/O address @16,21: over the General Purpose Interface Bus (GPIB). Primary address 16 selects the graphic device (an X-Y plotter for example). This primary address is preassigned to the graphic device through hardware connections. The secondary address 21 is issued to the graphic device by default and tells the device to prepare to receive the X and Y coordinates from a MOVE statement.

After the I/O address is issued, the BASIC interpreter converts the increments (64,78) to the coordinates of the appropriate graphic data point. These coordinates are then converted from user data units to graphic display units (GDUs) and sent to the specified graphic device over the GPIB. The X coordinate is sent first (most significant digit first) followed by the Y coordinate (most significant digit first). Once these values are received by the graphic device, the device moves its writing tool from its present position to the specified data point.

THE ROTATE STATEMENT

Syntax Form:

[Line number] ROT numeric expression

Descriptive Form:

[Line number] ROTATE rotation angle measured in the current trigonometric units

PURPOSE

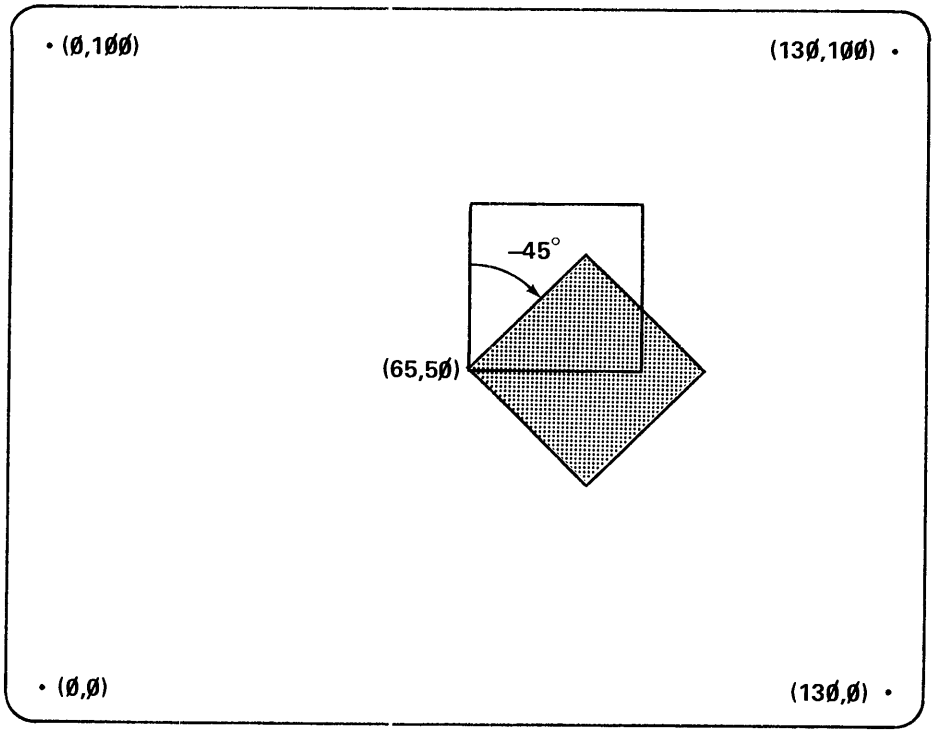
The ROTATE statement sets an environmental parameter which determines the rotation angle for the execution of subsequent RDRAW statements and RMOVE statements.

EXPLANATION

The following example illustrates how the rotation angle parameter effects a relative draw (RDRAW) and a relative move (RMOVE):

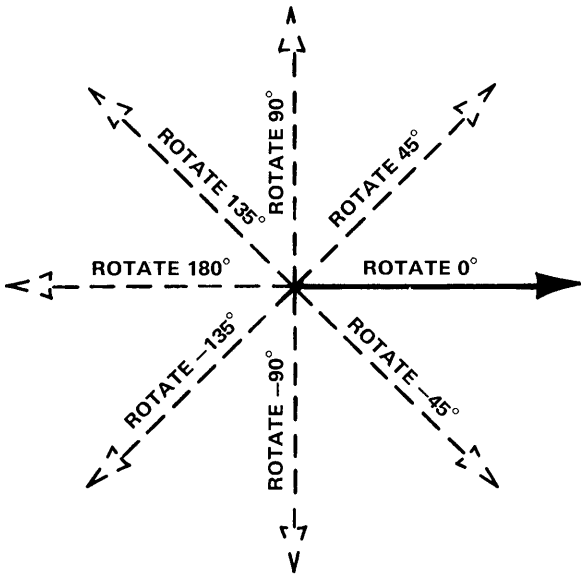
```
100 MOVE 65,50
200 RDRAW 30,0
300 RDRAW 0,30
400 RDRAW -30,0
500 RDRAW 0,-30
```

GS Display Output



This program causes the BASIC interpreter to draw a square. The unshaded square is drawn when the program is executed with the rotate parameter set equal to Ø (the default value). The rotate parameter is assumed to be measured in radians unless the trigonometric units are set to degrees or grads. (Refer to the SET statement in the Environmental Control section for a complete explanation of the trigonometric units setting.)

The same program draws the shaded square when the rotate parameter is set to –45 degrees. (The square is shaded for illustrative purposes only.) Notice that each vector is rotated –45° from its original position. The vectors are rotated around their points of origin using the Ø degree position as a reference. The following figure illustrates how vectors are rotated.



To set the rotation angle to one of the above, the appropriate statements are:

```
SET DEGREE
ROTATE X
```

The value of X can be either positive or negative and specifies the angle of rotation from the zero position. Positive values cause the vector to be rotated in a counterclockwise direction; negative values cause rotation in a clockwise direction. Once the rotation parameter is set, it remains set at that value until changed by another ROTATE statement, an INIT statement, or a system power up sequence.

THE SCALE STATEMENT

Syntax Form:

[Line number] SCA numeric expression , numeric expression

Descriptive Form:

[Line number] SCALE horizontal scale factor , vertical scale factor

Purpose

The SCALE statement specifies how many user data units are equivalent to a graphic display unit (GDU). The horizontal and vertical scale factors are specified independently. The position of the alphanumeric cursor at the time the statement is executed determines the point of origin (Ø,Ø).

Explanation

User Data Units vs Graphic Display Units

The term user data units refers to the unit of measure you choose to work with for a particular graphing application. Gallons, miles, minutes, pounds per square inch, dollars, francs, and megatons are examples of user data units. Normally, the WINDOW statement is used to define what the user data units are for a particular application.

Graphic display units, on the other hand, refer to the units of measure the BASIC interpreter uses internally to address points on the GS display or points on the graphic surface of an external peripheral device. Internally, the BASIC interpreter "sees" each graphic surface as a two dimensional plane which is divided into 1ØØ graphic display units on the shortest axis. This means the BASIC interpreter sees the GS display as 1ØØ GDUs high and 13Ø GDUs wide. Points on the display are addressed internally on this basis. Because GDUs can be specified with a decimal part, the BASIC interpreter can address almost an infinite number of points on the screen.

When coordinate values are specified in user data units in graphic statements like MOVE and DRAW, the BASIC interpreter converts the values to GDUs before addressing a point on the

screen. If the WINDOW and VIEWPORT parameters are set to their default values, a one-to-one relationship between user data units and graphic display units occurs and the conversion is not necessary. This one-to-one relationship is defined as the scale factor of 1.

Scale Factor Defined

The term "scale factor" refers specifically to the ratio between user data units and graphic display units on the GS display or on an external graphic surface. For example, the scale factor 2 means that two user data units are equivalent in length to one graphic display unit; the scale factor 1Ø means that 1Ø user data units are equivalent to one GDU, and so on.

Decimal scale factors are also possible. For example, the scale factor .1 means that one user data unit is equivalent to 1Ø graphic display units.

In general:

$$\text{Scale Factor} = \frac{\text{User Data Units}}{\text{Graphic Display Unit}}$$

Defining a Window with the SCALE Statement

The SCALE statement provides an alternate method of defining a window for a particular graphing application. This method uses scale factors to define the window instead of specifying the minimum and maximum values on each axis.

Two parameters must be specified in the SCALE statement. The first parameter specifies the horizontal (X) scale factor. The second parameter specifies the vertical (Y) scale factor. The position of the alphanumeric cursor at the time the SCALE statement is executed determines the point of origin (Ø,Ø).

A Practical Example

The best way to see how the SCALE statement works is through a practical example. Assume that you want to graph the function Y=X↑2 with the X limits set at ±1ØØ. Here's one way to go about it.

First, figure out what the scale factors should be on each axis. In this case, the value of X ranges from −1ØØ to +1ØØ for a total range of 2ØØ data units. Internally, the width of the GD display is 13Ø GDUs, so the appropriate scale factor for the horizontal axis is 2ØØ/13Ø. This is approximately equal to 1.538. Vertically, the maximum value to be graphed is 1ØØ↑2 which turns out to be 1ØØØØ. If the point of origin is spaced up ten GDUs from the bottom of the screen, the

graph looks better, so 90 GDUs are left from the point of origin to the top of the screen. To fit 10000 data units into this space, the appropriate scale factor is 10000/90 which is approximately 111.111. To establish these scale factors on the GS display, the appropriate SCALE statement is . . .

```
SCALE 1.538,111.111
```

or to be exact, the scale factors can be specified as numeric expressions:

```
SCALE 200/130,10000/90
```

(Specifying numeric expressions produces a scale factor with maximum decimal accuracy.)

Before the SCALE statement is executed, the alphanumeric cursor must be positioned to the desired point of origin. In this case, it's best to execute an INIT statement to set the WINDOW and VIEWPORT parameters to their default values, then execute a MOVE to the point where you want the point of origin. The appropriate statements are . . .

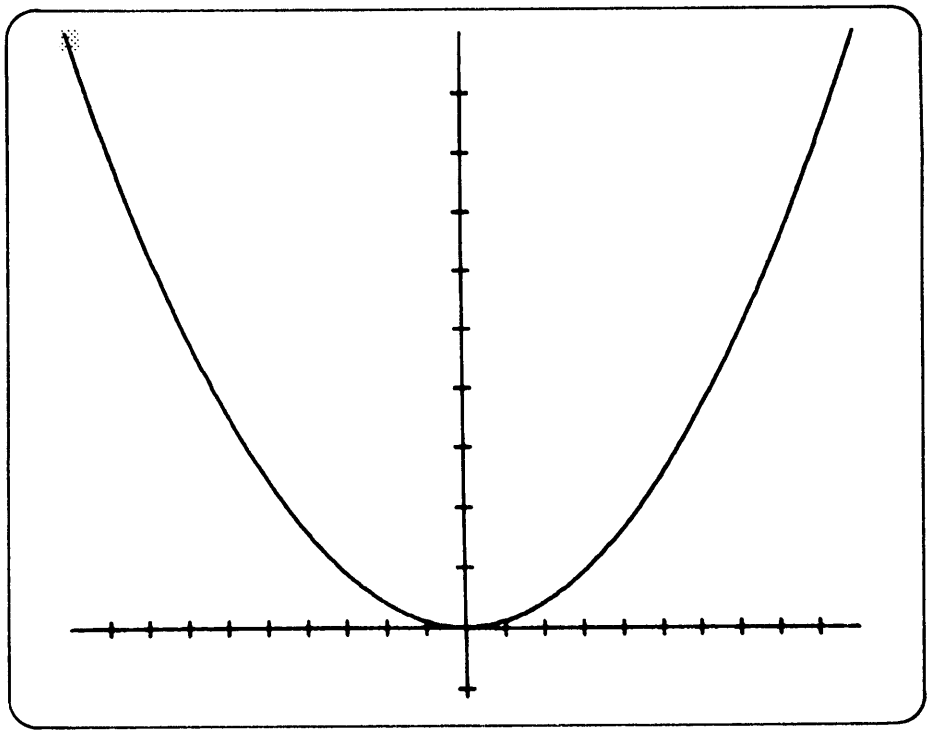
```
INIT
MOVE 65,10
```

Since the INIT statement establishes a one-to-one ratio between user data units and graphic display units, the statement MOVE 65,10 positions the cursor in the center of the screen, ten GDUs up from the bottom.

Here's the complete program which establishes the proper scale factors and draws the function Y=X↑2:

```
100 INIT
110 MOVE 65,10
120 SCALE 200/130,10000/90
130 REM The Following Routine Plots the Function X↑2
140 PAGE
150 AXIS 10,1000
160 MOVE -100,10000
170 FOR X=-100 TO 100
180 DRAW X,X↑2
190 NEXT X
200 HOME
210 END
```

GS Display Output



Notice that the AXIS statement in line 15Ø places tic marks on the horizontal axis every 1Ø units and tic marks on the vertical axis every 1ØØØ units. By counting the tic marks it can be seen that exactly 2ØØ data units fit into the viewport horizontally and 1ØØØØ data units fit into the space from the point of origin to the top of the viewport. The WINDOW statement with the parameters −1ØØ,1ØØ,−1111.1,1ØØØØ could have been specified in line 12Ø to produce the same result.

Reducing the Size of the Viewport

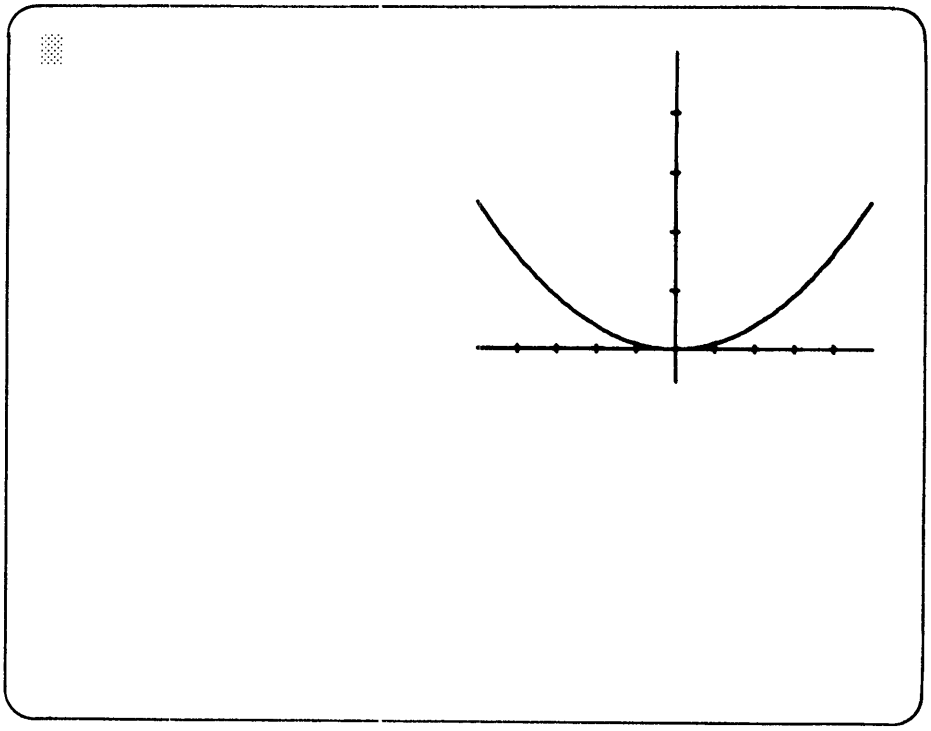
Reducing the size of the viewport does not change the scale factor.

For example, if the statement . . .

```
105 VIEWPORT 65,130,50,100
```

is inserted into the previous program, the following results are obtained:

GS Display Output



Notice that the viewport is defined as the upper right corner of the screen and the plot is restricted to this area; the scale factors, however, are not reduced proportionally. In this case, the dimensions of the viewport are reduced by one-half, so the numeric range on each axis is reduced by one-half. The scale factors (user data units per GDU) remain the same.

THE USER DATA UNIT CONCEPT

Introduction

The beauty of the Graphic System is that it allows you to work in your own units of measure. The term "user data units" refers specifically to the units of measure you select for a particular graphing application. If you're a salesman and want to draw a sales graph, for example, you'll want to work with "dollars" on the vertical axis and "months" or "years" on the horizontal axis. If you're a physics student, you might want temperature in degrees centigrade on the vertical axis and pounds per square inch on the horizontal axis. Whatever the application, the Graphic System allows you to specify the units of measure you want to work with, and for the duration of the application, you can "talk" to the BASIC interpreter in those units.

A Practical Application

Here's a real life example to help explain the user data units concept. Suppose you decide to start jogging and you want the Graphic System to keep track of your progress over a period of time. You can program the system to draw a graph which plots miles jogged per month, miles jogged per week, or even miles jogged per day. If you're jogging in Europe, you'll probably want to graph kilometers jogged per day instead of miles jogged per day. Whatever the case, you establish the appropriate units of measure on the X and Y axis with the WINDOW statement, and then specify all your graph coordinates in those units for graphic statements like MOVE, RMOVE, DRAW, and RDRAW.

Graphing Miles Jogged Per Week

Assume you collect the following data during the first week of jogging:

Monday — 3 miles
Tuesday — Ø miles (didn't run because you overdid it on Monday)
Wednesday — 1 mile
Thursday — 1/2 mile
Friday — Ø miles
Saturday — 2miles
Sunday — 3/4 miles

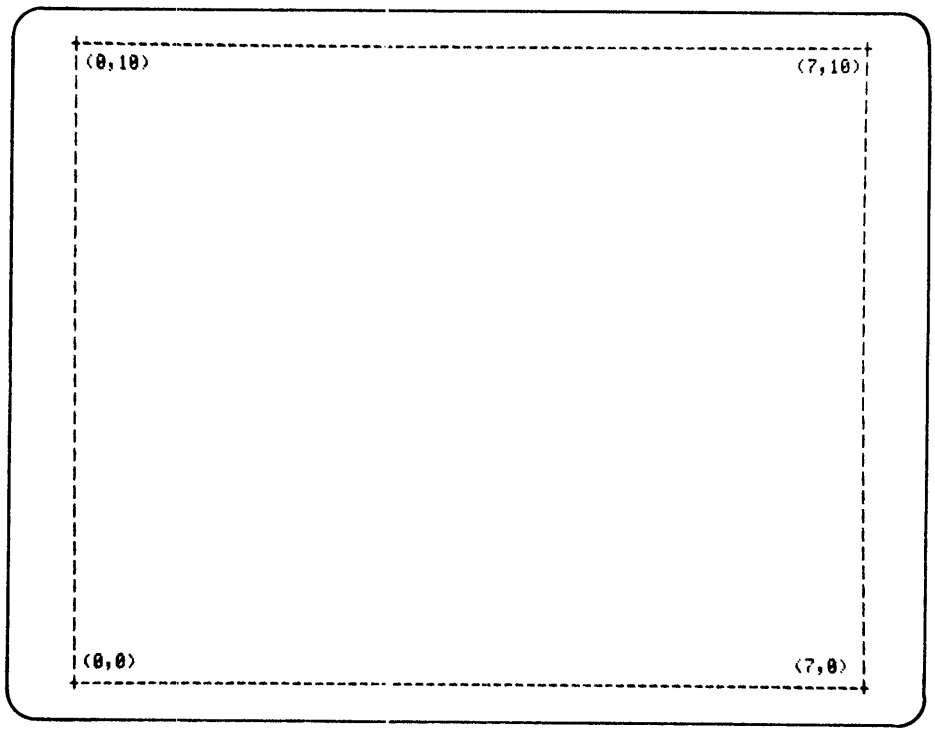
In order to draw the graph using this data, the first step is to establish the drawing boundaries on the GS display. The area within the drawing boundary is called the "viewport." The VIEWPORT statement selects a viewport which is smaller than the maximum screen area. For now though, let's stick with the maximum sized viewport which is automatically established on system power up and after the execution of an INIT statement. Later we'll change the viewport to a smaller size to show you how it's done.

We're plotting time (in days) against distance (in miles). These units of measure are established by executing a WINDOW statement. Enter the keyword WINDOW followed by the horizontal value on the left side of the screen, the horizontal value on the right side of the screen, the vertical value at the bottom of the screen, and the vertical value at the top of the screen. These values can be negative or positive and must be within the range ±8.988465674E+ 307 (the numeric range for the system). For this example, let's divide the viewport into seven increments horizontally (one for each day of the week) and ten increments vertically (one for each mile). The appropriate statement is . . .

```
WINDOW Ø, 7, Ø, 1Ø
```

Enter the statement and press RETURN. The results are shown below:

GS Display Output



The boundaries of the default viewport are shown in dashed lines because they aren't actually drawn on the screen; neither are the numbers you just specified in the WINDOW statement. After the WINDOW statement is executed, it's up to you to remember what units you established. The BASIC interpreter also remembers, but it doesn't know what the units

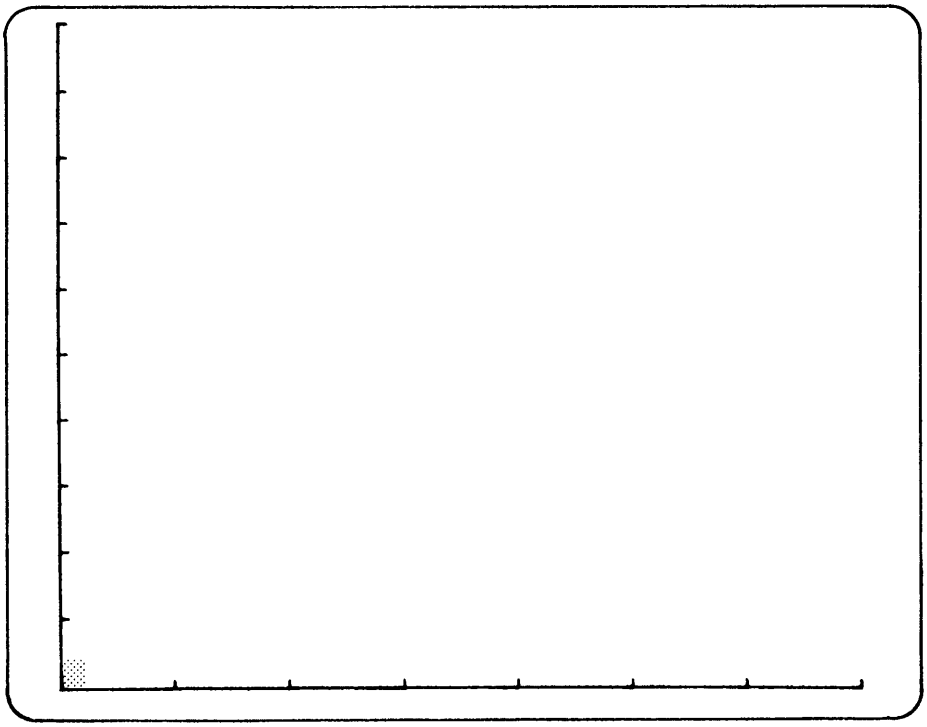
represent. At this point, it's up to you again to keep track of the fact that the horizontal units represent "days" and the vertical units represent "miles".

You can take advantage of the AXIS statement now to help you remember what the units are on each axis. Enter the following statement and press RETURN:

```
AXIS 1,1
```

This statement draws the axes shown below:

GS Display Output



The parameters 1,1 tell the BASIC interpreter to draw an X-Y axis with tic marks at one unit intervals. You can see that there are seven tic marks on the horizontal axis (one for each day) and ten tic marks on the vertical axis (one for each mile). That's all we'll do with the AXIS statement for now. A complete explanation of the AXIS statement is given at the beginning of this section.

Now, let's draw the graph. Since the AXIS statement leaves the alphanumeric cursor at the point of origin (the place where the axes intersect) we don't have to execute a MOVE Ø,Ø statement to get there. Normally, moving the cursor to the starting point on the graph is required as the first step.

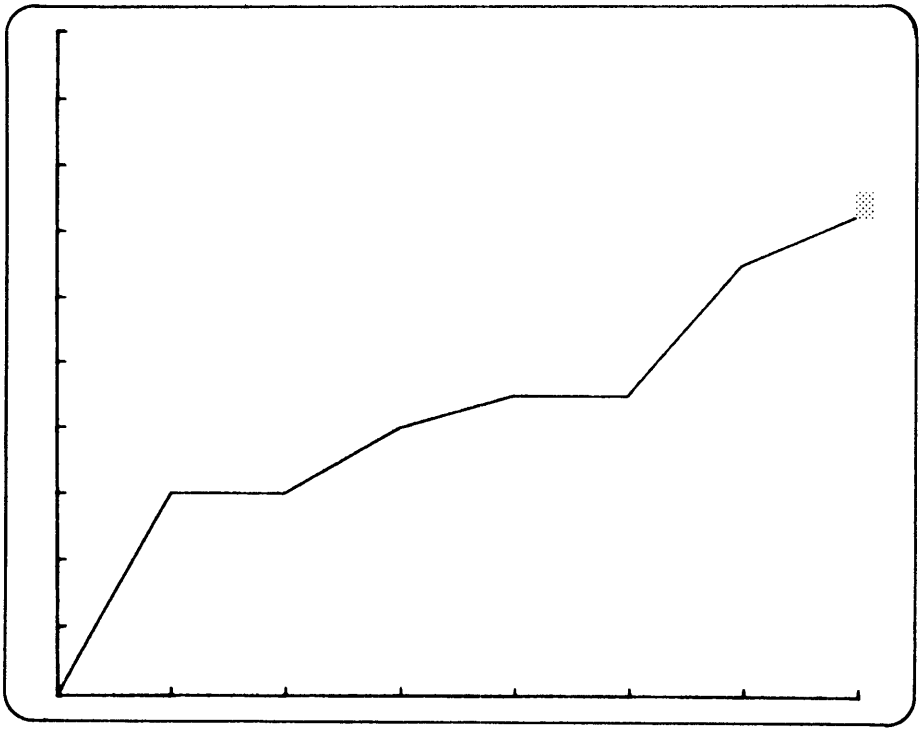
On Monday, the first day, you jogged 3 miles, so the appropriate statement is DRAW 1,3. This tells the BASIC interpreter to draw a line from the present position of the cursor to the coordinates 1,3. Notice that coordinates are specified in user data units, the units you established with the WINDOW statement. The 1 means "draw horizontally to the first day"; the 3 means "go to the three mile position vertically".

On Tuesday, you didn't run because of soreness. The data says Ø miles, but we want to keep track of the total mileage. Three plus zero is three, so the next DRAW statement is DRAW 2,3. This draws a line from the total miles jogged on Monday to the total miles jogged up to Tuesday. The Wednesday coordinates are specified DRAW 3,4, and so on.

You might have discovered by now that the graph can't be drawn one statement at a time from the GS keyboard because the entry statements get in the way. Here's a complete BASIC program which draws the graph under program control. Enter the program from the GS keyboard, type RUN, and press RETURN.

```
100 INIT
110 WINDOW Ø,7,Ø,10
120 PAGE
130 AXIS 1,1
140 DRAW 1,3
150 DRAW 2,3
160 DRAW 3,4
170 DRAW 4,4.5
180 DRAW 5,4.5
190 DRAW 6,6.5
200 DRAW 7,7.25
210 END
```

GS Display Output



Notice that only lines are drawn. It's up to you to label the graph with alphanumerics via the PRINT statement; or just keep the units of measure in your head. (Refer to the manual PLOT 50: Introduction to Graphics Programming in BASIC for an explanation on how to label graphs.)

Changing the Size and Location of the Viewport

So far we've used the whole screen to plot the graph. Now let's reduce the drawing area to the upper-right corner of the screen and see what happens.

The BASIC interpreter "talks" to the GS display internally using a unit of measure called a GDU (Graphic Display Unit). Internally, the BASIC interpreter "sees" the GS display as 130 GDUs horizontally and 100 GDUs vertically. All internal communications with the GS display are carried on in GDUs. This means that when we "talk" to the BASIC interpreter in our own user data units, the BASIC interpreter checks to see what the WINDOW statement parameters are, then converts the values to the appropriate GDU values. The BASIC interpreter uses the GDU values to address points on the screen. This conversion process is called a "transformation" and is automatically carried out by the BASIC interpreter when coordinate values are specified in graphic statements like MOVE, RMOVE, DRAW, and RDRAW.

The only time you need to think about GDUs is when you specify the drawing boundaries on the GS display or on an external peripheral device. We're going to do this next. The boundaries are established with the VIEWPORT statement. As stated before, the screen is internally divided into 130 GDUs on the horizontal axis and 100 GDUs on the vertical axis; the VIEWPORT parameters are specified on that basis. To define the drawing boundaries, the keyword VIEWPORT is executed, the first parameter marks the left boundary; the second parameter marks the right boundary; the third parameter marks the bottom boundary; and the fourth parameter marks the top boundary. All parameters are specified in GDUs.

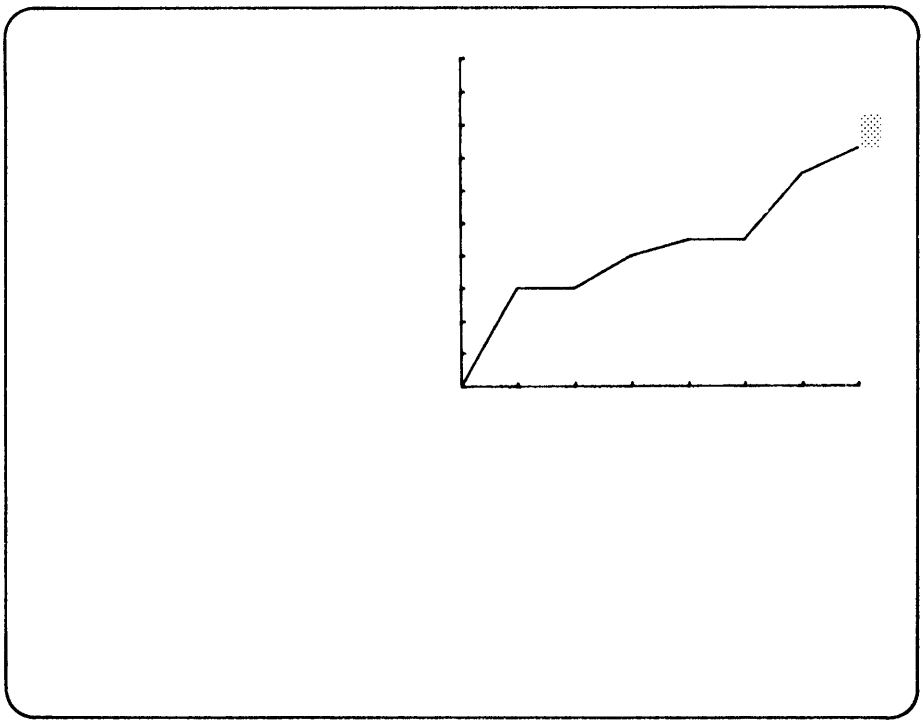
The following statement limits the drawing area to the upper-right corner of the screen:

```
VIEWPORT 65,130,50,100
```

Now, change line 100 in the previous program to line 100 as shown in the listing below. Run the program again and see what happens.

```
100 VIEWPORT 65,130,50,100
110 WINDOW 0,7,0,10
120 PAGE
130 AXIS 1,1
140 DRAW 1,3
150 DRAW 2,3
160 DRAW 3,4
170 DRAW 4,4.5
180 DRAW 5,4.5
190 DRAW 6,6.5
200 DRAW 7,7.25
210 END
```

GS Display Output



Notice that the graph is unchanged other than the fact that it is scaled down to fit the smaller viewport (upper-right corner).

Summary

This example illustrates how easy it is to draw a graph on the Graphic System. It serves only as an introduction to the rest of the topics in this section. It is important that you understand and remember the following points:

1. The term "user data units" refers to the units of measure you select for a particular graphing application.

2. You use the WINDOW statement to tell the BASIC interpreter what units of measure you want to work with.

3. The drawing boundaries on the GS display are established with the VIEWPORT statement. The VIEWPORT parameters are specified in graphic display units (GDUs) which are internal units used to address points on the GS display.

4. Once the viewport boundaries are established, you forget about GDUs. The coordinate values you specify in graphic statements like MOVE and DRAW are specified in user data units.

Refer now to the topics in the rest of this section for a detailed discussion on each graphic statement. Refer also to the PLOT 50 Introduction to Graphics Programming in BASIC manual for discussions on graphic programming concepts.

THE VIEWPORT STATEMENT

Syntax Form:

[Line number] VIE numeric expression , numeric expression , numeric expression
, numeric expression

Descriptive Form:

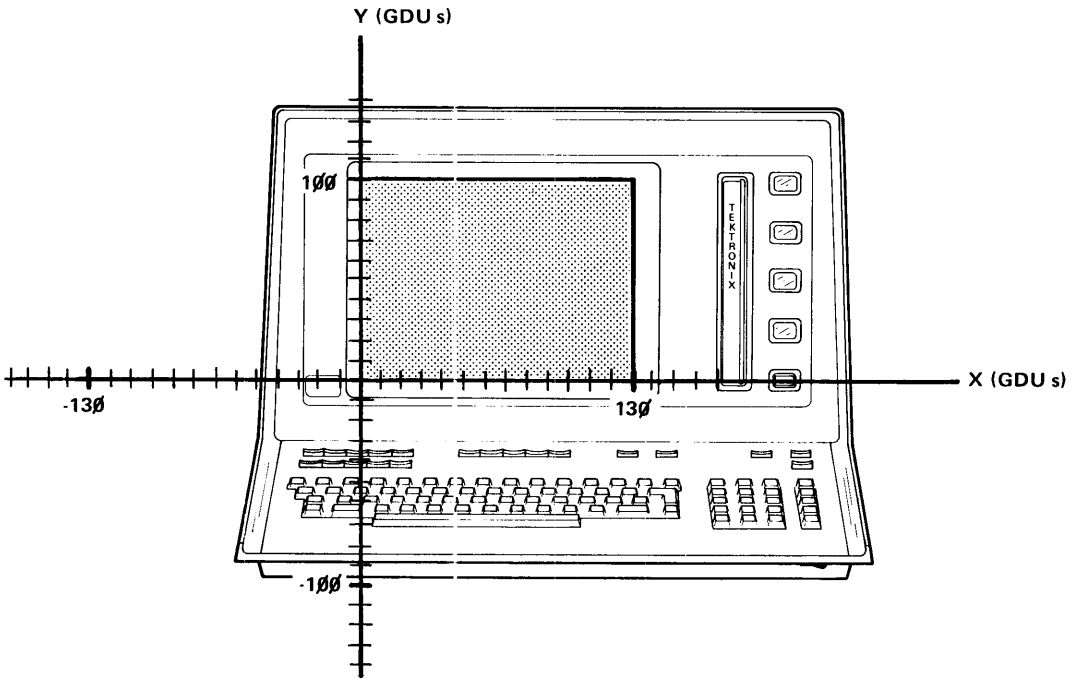
[Line number] VIEWPORT minimum horizontal value in GDU s , maximum
horizontal value in GDU s , minimum vertical value in GDU s ,
maximum vertical value in GDU s

Purpose

The VIEWPORT statement defines the boundaries of the drawing surface on the GS display. The VIEWPORT parameters are automatically set to Ø,13Ø,Ø,1ØØ by default on system power up and after the execution of an INIT statement.

Explanation

Internally, the GS display represents the first quadrant in a Cartesian Coordinate System as shown in the following illustration. Both axes are marked in graphic display units (GDUs) which are internal units of measure for all graphic commands. It can be seen that the display screen covers an area 13Ø GDUs wide and 1ØØ GDUs high. This area defines the maximum drawing surface available on the display and is shaded for illustrative purposes.



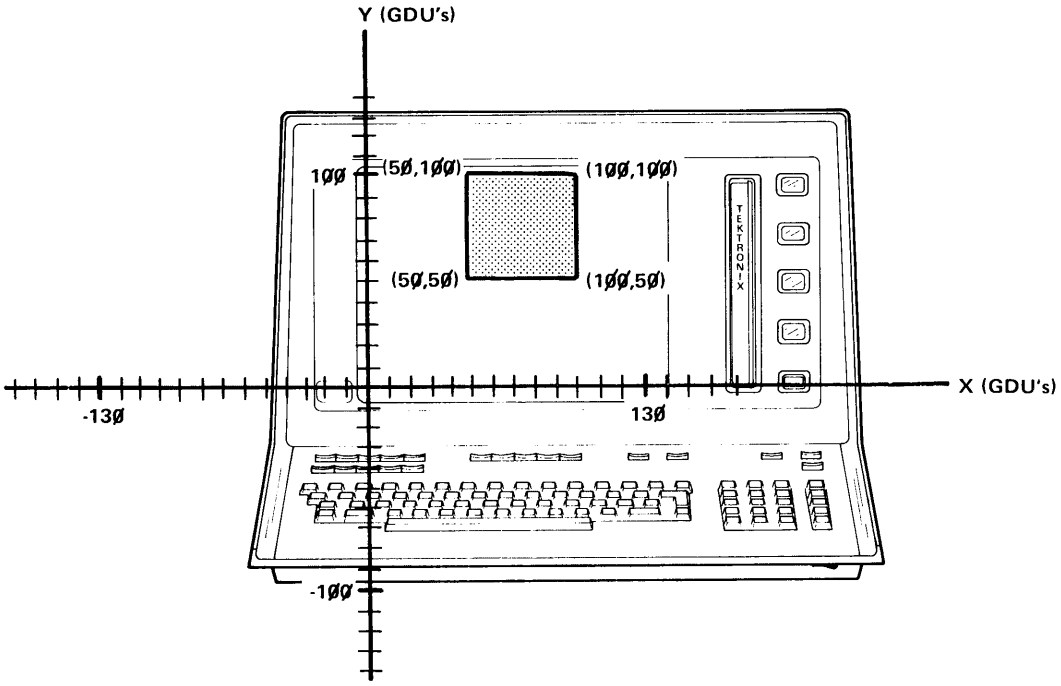
The VIEWPORT statement controls the boundaries of the drawing surface. These boundaries are specified as follows: minimum horizontal (X) value in GDUs, maximum horizontal (X) value in GDUs, minimum vertical (Y) value in GDUs, maximum vertical (Y) value in GDUs.

The VIEWPORT parameters are automatically set to Ø,13Ø,Ø,1ØØ by default on system power up and after the execution of an INIT statement.

Specifying a Smaller Viewport

Sometimes it's necessary to reduce the size of the drawing surface to leave room for printed messages. This is done by changing the VIEWPORT parameters. For example:

```
VIEWPORT 5Ø,1ØØ,5Ø,1ØØ
```



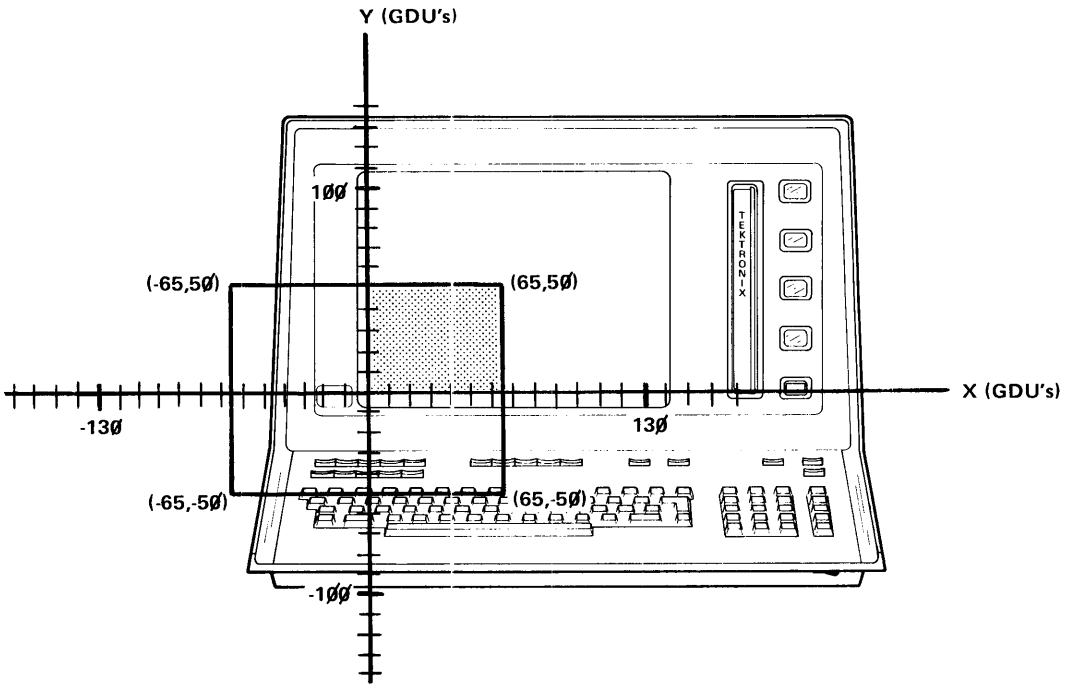
This VIEWPORT statement reduces the drawing surface to an area as shown by the shaded portion of the diagram above. All graphic output is automatically scaled to fit this area. The viewport can be changed as many times as desired throughout the course of a program. This allows the BASIC interpreter to draw several plots in different areas on the screen.

The viewport can be defined to be any shape or size; it can be tall and skinny or short and fat. Whatever the size, all graphic output occurs within the defined area. A zero width or height is not allowed.

Specifying a Viewport Larger than the Screen

Specifying a viewport larger than the size of the display can be done, however, it's not recommended. The following example illustrates what happens when a portion of the viewport is defined off the screen. The parameters are set as follows:

```
VIEWPORT -65,+65,-5Ø,+5Ø
```



As you can see, only the upper-right corner of the viewport lies on a physical drawing surface. Graphics output can be displayed in this area, however, any attempt to MOVE or DRAW to a coordinate which is off the screen results in a total scissor. That is, the MOVE or DRAW is not executed and the cursor remains in its present position. This has an over all effect of distorting the plot. It is good practice never to define the viewport outside the default parameters.

THE WINDOW STATEMENT

Syntax Form:

[Line number] WIN numeric expression , numeric expression , numeric expression
, numeric expression

Descriptive Form:

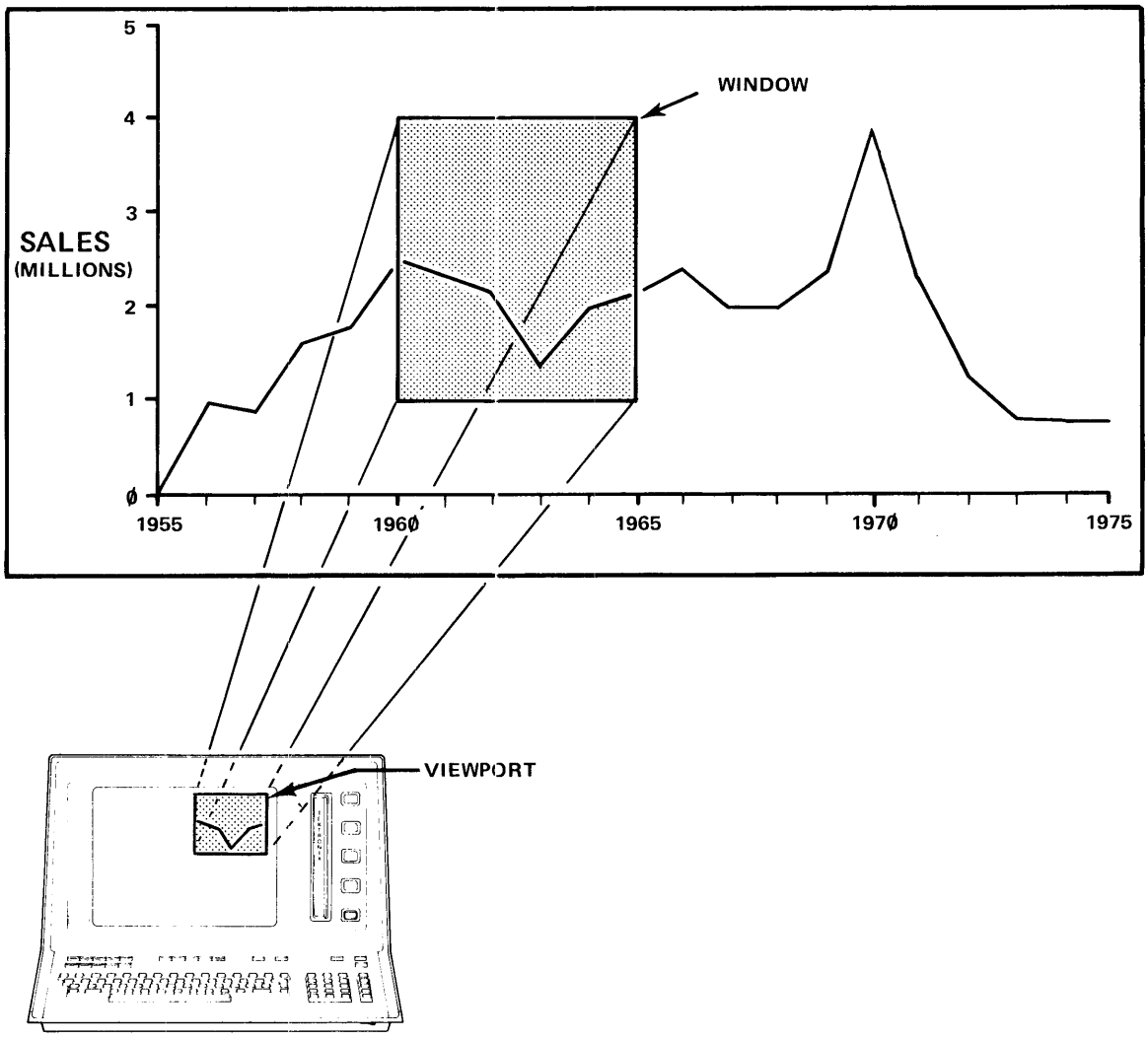
[Line number] WINDOW minimum horizontal (X) value in user data units , maximum horizontal (X) value in user data units , minimum vertical (Y) value in user data units , maximum vertical (Y) value in user data units

Purpose

The WINDOW statement specifies how many user data units fit inside the viewport. The user data units can be any units of measure you wish to work with (inches, miles, dollars, years, etc.). The WINDOW parameters are automatically set to Ø,13Ø,Ø,1ØØ by default on system power up and after the execution of an INIT statement.

Explanation

The "window" is defined as the image of the viewport cast onto the user data space. Refer to the following figure.



In this example, a portion of a sales graph is mapped into a viewport which is defined in the upper-right corner of the screen. The appropriate statements are as follows:

```
VIEWPORT 65,13Ø,5Ø,1ØØ
WINDOW 196Ø,1965,1,4
```

Notice that the viewport parameters are specified in graphic display units. This defines the physical size and location of the drawing area. The viewport is then marked up in user data units with the WINDOW statement. In this case, the width of the viewport starts at the year 196Ø and ends in the year 1965. The vertical axis of the viewport is labeled in millions of dollars starting at 1 million and extending to 4 million.

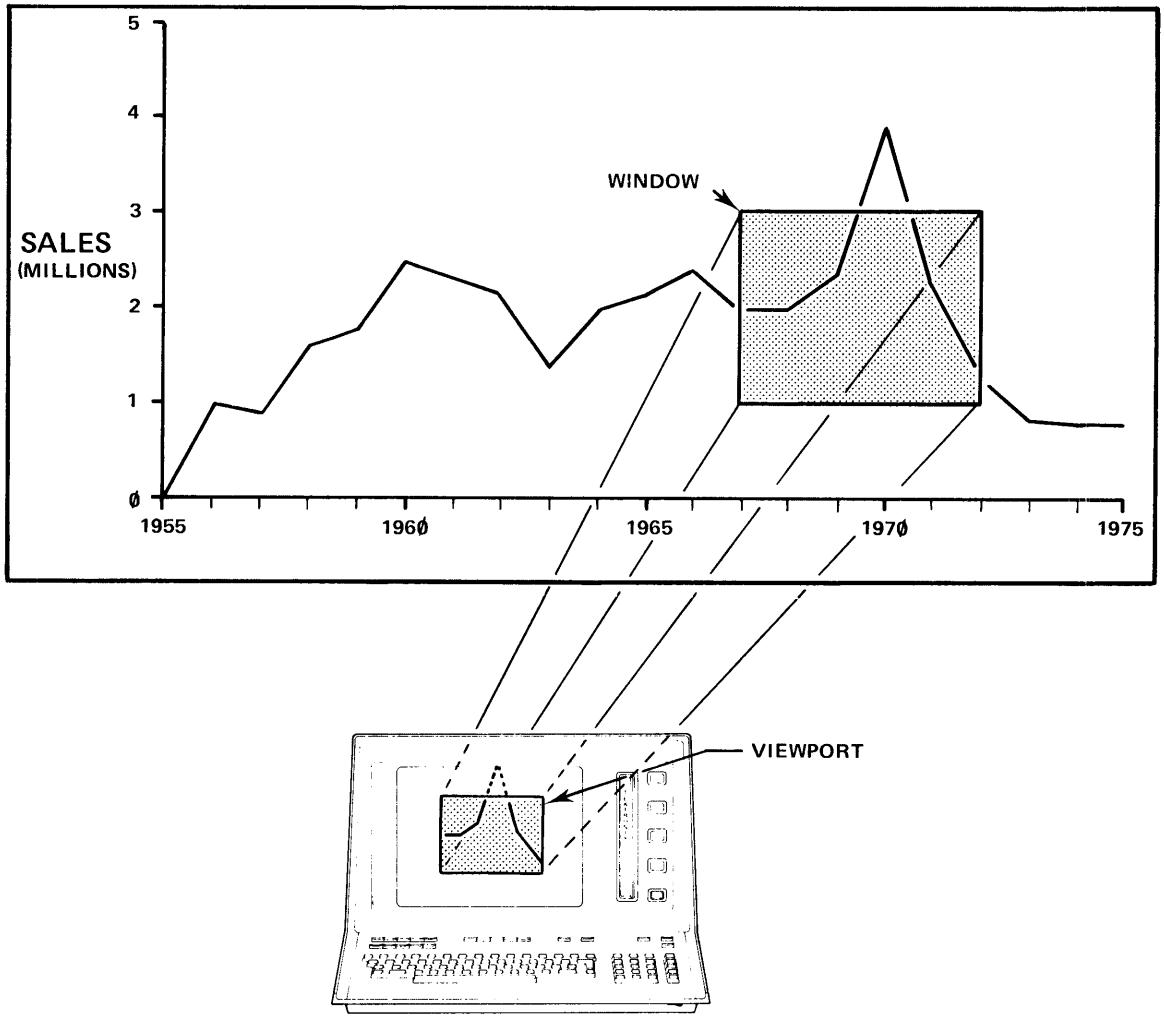
Now, all you have to do is specify the year and the amount of sales dollars in a MOVE or DRAW statement. For example, to draw a portion of the graph, you might specify the following:

```
360 DRAW 1963,1.4
```

This indicates that the sales are 1.4 million dollars in the year 1963 and the BASIC interpreter draws a line to the coordinates (1963,1.4). The line starts at the present position of the cursor (the 1962 sales figure).

Clipping

If the parameters of a MOVE, RMOVE, DRAW, or RDRAW statement specify a destination point which is outside the window, the BASIC interpreter executes a theoretical move or draw to that point. If a DRAW is executed, then only that portion of the vector which lies inside the window is drawn. The portion of the vector which lies outside the window is clipped (chopped off) at the edge of the viewport. The following figure illustrates clipping action:



In this example, the viewport is defined as an area in the middle of the screen and the window covers a section of the sales graph on the right. The appropriate statements to set this up are:

```
VIEWPORT 35,95,3Ø,7Ø
WINDOW 1967,1972,1,3
```

Because the vertical axes of the window only extend to 3 million dollars, the sales figure for 197Ø is clipped. Notice that the BASIC interpreter goes through the motions of drawing the complete graph even though the portion of the graph which lies outside the window is not drawn.

CHARACTER STRINGS

Section 10

CHARACTER STRINGS

INTRODUCTION TO CHARACTER STRINGS

The term "character string" refers to any sequence of letters, numbers, or symbols enclosed in quotation marks. Character strings are also referred to as strings, literal strings, and string constants. Usually, a character string represents a piece of written text or a message to be printed on the GS display or an external peripheral device.

Four keywords can be used to enter character strings into memory and eight string functions are available to manipulate character strings once they are in memory. Each string constant entered into memory must be assigned to a string variable. Both string constants and string variables can be arranged into string expressions using the concatenation operator (&). No provision is made for string arrays.

Dimensioning String Variables

As stated above, every string constant must be assigned to a string variable when it is entered into memory (unless otherwise specified). String variables have a maximum working size of 72 characters by default. Sometimes it is necessary to increase the maximum working size of a string variable to accommodate a string with more than 72 characters; and, if a string contains less than 72 characters, it is good practice to reduce the working size to conserve memory space. The working size of a string variable is changed with the DIM statement.

Assigning String Constants to String Variables

String constants are assigned to string variables in several ways. If the assignment is made directly within the BASIC program, the LET statement must be used. String constants assigned in this manner must be enclosed in quotation marks.

String constants are assigned to string variables as a program executes with the INPUT and READ statements.

The INPUT statement allows a string constant to be assigned to a string variable from the GS keyboard, the internal magnetic tape unit, or an external peripheral device on the General Purpose Interface Bus. If the GS keyboard is specified as the input source, the program waits for the keyboard operator to enter a character string into the line buffer and press the RETURN key; the BASIC interpreter then assigns the entry to the specified string variable and program execution continues. If the internal magnetic tape unit or an external peripheral device is specified as the input source, the BASIC interpreter inputs a character string from the specified device.

The READ statement retrieves character strings formatted in machine dependent binary code and assigns the strings to the specified string variables. Normally, the character strings are first output to a peripheral device with the WRITE statement before they are brought back into memory with the READ statement. If a peripheral device is not specified, then string constants are taken from the current DATA statement and assigned to the specified string variables.

String Functions

Once character strings are entered into the memory, they can be manipulated and changed using eight different string functions. The following is a brief description of each string function:

String Function	Purpose
LEN	The LEN function returns the number of characters in the specified character string.
POS	The POS function searches for and returns the position of the first occurrence of the specified substring within the specified string.
SEG	The SEG function locates the specified substring within the specified string and assigns the substring to the specified string variable. The original string is not altered or changed.
REP	The REP function deletes the specified characters from the specified string and replaces those characters with a specified substring.
VAL	The VAL function converts a number which is expressed as a character string into a number which can be used as numeric data in math operations.
STR	The STR function converts the specified numeric expression to an ASCII character string.
ASC	The ASC function returns the decimal number corresponding to the specified ASCII character.
CHR	The CHR function returns the ASCII character equivalent of the specified numeric expression.

THE ASC FUNCTION

Syntax Form:

ASC { string constant / string variable }

Purpose

The ASC (ASCII Character) function returns a decimal number corresponding to the specified ASCII character. This function is the inverse of the CHR function.

Explanation

The ASC (ASCII Character) function is a monadic function which requires a string constant or a string variable as a parameter. The result of the function is a decimal number within the range Ø to 255, inclusive. This number corresponds to the decimal value assigned to the ASCII character. (See the ASCII Character Value Chart in Appendix B.) If the parameter of the ASC function contains more than one character, then the decimal value of the first character is returned; all other characters are ignored. For example:

Statement	Result
ASC "A"	65
ASC "AB"	65
ASC """"	34
32Ø LET J = ASC "*"	J = 42

NOTE: In the third example, two quotation marks ("") are used to represent one quotation mark (") inside a string. The outside quotation marks are used as delimiters to mark the beginning and ending of the string.

THE CHR FUNCTION

Syntax Form:

[Line number] [LET] string variable = CHR (numeric expression)

Purpose

The CHR (Character) function returns the ASCII character equivalent of the specified decimal number and assigns the character to the specified string variable. This function is the inverse of the ASC function.

Explanation

The parameter of the CHR function is a numeric expression which is reduced to a numeric constant and rounded to an integer. The integer must fall within the range Ø to 127, inclusive, or an error occurs. Once the parameter is reduced to an integer, the ASCII character which corresponds to the decimal number is assigned to the specified string variable. For example:

Statement	Result
A$=CHR(65)	A$ = "A"
B$=CHR(127)	B$ = "↓"
C$=CHR(122)	C$ = "z"

If the decimal value of an ASCII control character is specified, then the control character is assigned to the string variable; however, there is no printed indication. If the string variable is specified as a parameter in an output statement, then the actual control character is sent to the output device. This may cause the output device to execute a control function.

For example, if CHR (13) is assigned to A$, and A$ is specified in a PRINT statement, then a Carriage Return character is sent to the GS display and a Carriage Return/Line Feed is executed.

A complete list of ASCII control characters and their decimal value equivalent can be found in the ASCII Character Value Chart in Appendix B.

THE DIM STATEMENT

Modified Syntax Form:

[Line number] DIM string variable (numeric expression) [, string variable (numeric expression)] . . .

Purpose

The DIM (Dimension) statement establishes the maximum working size of one or more string variables. If a string variable is not dimensioned in a DIM statement, then the maximum working size of the string variable is set at 72 characters by default.

Explanation

The following example illustrates how a string variable is dimensioned:

```
250 DIM Q$(5)
```

When this statement is executed, the maximum working size of Q$ is set to 5 characters. If an attempt is made to assign more than 5 characters to Q$ with an assignment statement, an error results and program execution is aborted. If an attempt is made to assign more than 5 characters to Q$ with an INPUT or READ statement after this statement is executed, then the first 5 characters are retained; all other characters are lost and the program continues executing.

Several string variables can be dimensioned in one DIM statement if the variables are separated by commas as shown below:

```
440 DIM A$(30), B$(X),C$(X12+Y)
```

Notice also in this statement that the dimensioned size of a string variable can be specified as a numeric expression. The only requirement is that all variables in the numeric expression must have assigned values by the time the DIM statement is executed; otherwise, an error results and program execution is aborted. When a numeric expression is specified, the BASIC interpreter reduces the expression to a numeric constant and rounds the numeric constant to a positive integer.

All string variables may be dimensioned more than once without deleting them; however, the total number of characters specified must be less than or equal to the number specified the first time the variable is dimensioned. If a string variable is dimensioned to a larger size, then the variable must first be deleted from memory with the DELETE statement and redimensioned to a larger size with the DIM statement. If a string variable is deleted, the dimensioned size for that variable defaults back to 72 if the variable is used again without specifying its working size with the DIM statement.

See the Language Elements section for a complete explanation of the DIM statement.

THE INPUT STATEMENT

Modified Syntax Form:

[Line number] INP [I/O address] string variable [, string variable]...

Purpose

The INPUT statement assigns character strings to string variables from an internal or external peripheral device. Incoming character strings are assumed to be formatted in ASCII code.

Explanation

Input from the GS Keyboard

The INPUT statement is used to assign character strings to specified string variables from the GS keyboard. Usually, the GS keyboard entries are answers to questions which are printed on the display. For example:

```
200 PRINT "What is your name ?"
210 INPUT V$
```

When line 200 in this example is executed, the BASIC interpreter prints the message "What is your Name ?" on the GS display. Line 210 then causes the BASIC interpreter to display a blinking question mark in place of the alphanumeric cursor and wait for an entry from the GS keyboard. Up to 72 characters can be entered. As each character is entered, the character is printed on the GS display and the blinking question mark moves one place to the right. The entry is terminated when the RETURN key is pressed.

The keyboard entry does not have to be enclosed in quotation marks; in fact, if quotation marks are entered, the BASIC interpreter considers the quotes as part of the string.

If more than one string variable is specified in the INPUT statement, (INPUT A$,B$ for example), then all the characters entered up to the first Carriage Return are assigned to the first variable and all of the characters entered up to the second Carriage Return are assigned to the second variable. The BASIC interpreter continues to display the blinking question mark until every variable specified in the INPUT statement has an assigned value. If the RETURN key is pressed without entering any characters from the keyboard, then an empty (or null) string is assigned to the specified string variable.

Inputting Character Strings from the Internal Magnetic Tape Unit

If the I/O address @33: is specified in the INPUT statement, then the BASIC interpreter inputs character strings from the internal magnetic tape unit and assigns the character strings to the specified string variables. For example:

```
34Ø INPUT @33:R$
```

This statement causes the BASIC interpreter to input characters starting at the present position in the current file and assigns the characters to the specified string variable (R$). The characters are normally stored in ASCII code. The operation is terminated when a Carriage Return character is received from the magnetic tape; this marks the end of a logical record. If more than one string variable is specified, then the magnetic tape unit keeps sending characters until all the specified string variables have assigned values. Carriage Returns are the only valid delimiters for strings. (Refer to the Input/Output Operations section for complete information on accessing magnetic tape files.)

Input from an External Peripheral Device

Inputting character strings from an external peripheral device is the same as inputting character strings from the internal magnetic tape unit, except that a different primary address is specified. If the peripheral device is on the General Purpose Interface Bus, then a primary address from 1 to 3Ø is specified. If the peripheral device is connected to the optional Data Communications Interface then primary address 4Ø is specified. For example:

```
46Ø INPUT @16: X$,Y$,Z$
47Ø INPUT @4Ø: K$
```

Line 46Ø causes the BASIC interpreter to input three character strings from device 16 on the General Purpose Interface Bus and assign the character strings to the string variables X$,Y$, and Z$. Line number 47Ø causes the BASIC interpreter to input characters from the optional Data Communications Interface until a Carriage Return character is received. The characters are assigned to K$.

Refer to the Input/Output Operations section for complete information on the INPUT statement.

THE LEN FUNCTION

Syntax Form:

LEN { string constant / string variable }

Purpose

The LEN (Length of String) function returns the number of characters in the specified character string.

Explanation

The LEN function is a monadic function which requires a string variable or a string constant as a parameter. The result is an integer which represents the number of characters in the string (logical length) and not the dimensioned length. This function allows the program to check the actual number of characters in a string at any given time. For example:

```
LEN A$
  12
```

In this example, the keyboard operator asks for the current length of the character string assigned to A$. The BASIC interpreter returns 12 which indicates there are twelve characters in the string. The LEN function returns a numeric result and can be part of the numeric expression. For example, the following statements are valid statements:

```
230 LET X=3+(LEN G$)/2
240 IF LEN A$=5 THEN 500
```

If a string constant is specified as the parameter of the LEN function, then the string constant must be enclosed in quotation marks. For example:

```
680 M = LEN "SNAKE"
```

The numeric constant 5 is assigned to the variable M when this statement is executed.

THE LET STATEMENT AND THE CONCATENATION OPERATOR

Modified Syntax Form:

[Line number] [LET] string variable = string expression

Purpose

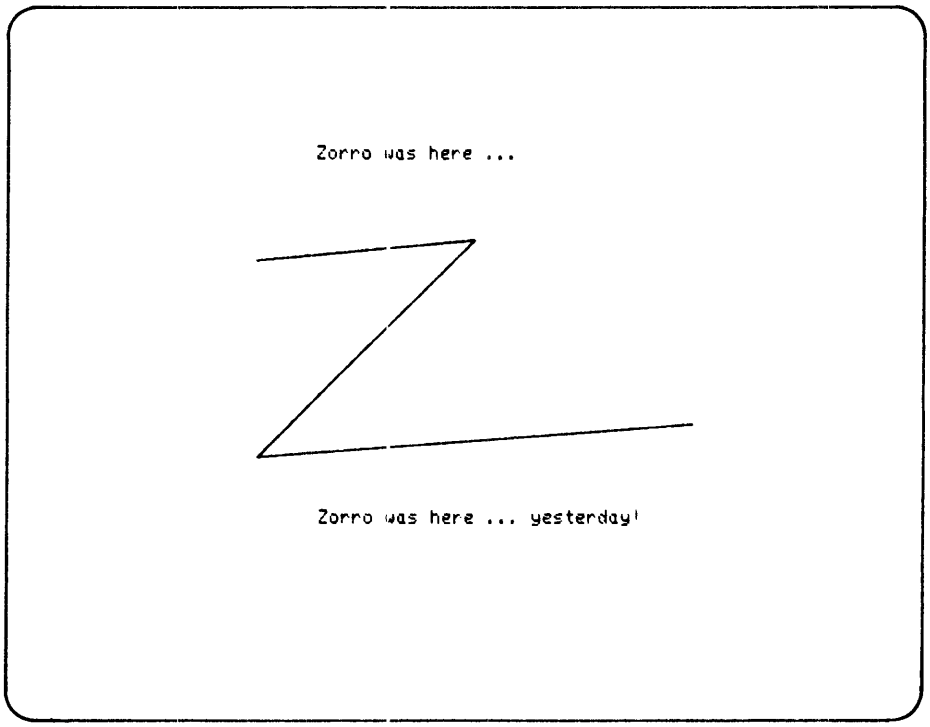
The LET statement assigns the specified string expression to the specified string variable.

Explanation

The LET statement is used in a BASIC program to assign the result of a string expression to a string variable. When the string expression is evaluated, each string variable in the expression must be defined, or an error occurs and program execution is aborted. The following statements illustrate several ways to assign character strings to string variables using the LET statement:

```
100 INIT
110 PAGE
120 LET A$="Zorro"
130 LET B$=" was here ..."
140 LET C$=A$&B$
150 MOVE 40,75
160 PRINT C$
170 LET D$=C$&" yesterday!"
180 MOVE 40,20
190 PRINT D$
200 MOVE 30,60
210 DRAW 65,63
220 DRAW 30,30
230 DRAW 100,35
240 HOME
250 END
```

GS Display Output



When line 1ØØ is executed, the system environmental parameters are initialized. This clears every value previously assigned to a variable. Line 11Ø pages the screen, then line 12Ø assigns the string constant "Zorro" to the string variable A$. Line 13Ø is executed next and the string constant " was here ..." is assigned to B$.

Line 14Ø illustrates how to concatenate two character strings in an assignment statement. When line 14Ø is executed, the string constant assigned to B$ is joined to the end of the string constant assigned to A$ and the result is assigned to C$. A move to the coordinates 4Ø,75 is executed in line 15Ø and the string assigned to C$ is printed on the GS display.

Line 17Ø illustrates how to concatenate a string constant to a character string which is assigned to a string variable. The resultant character string is assigned to D$. A move to the coordinates 4Ø,2Ø is executed in line 18Ø and the character string assigned to D$ is printed on the GS display.

Lines 2ØØ through 23Ø execute a series of moves and draws to create a "Z" on the GS display and the program terminates. The results are shown in the illustration above.

THE POS FUNCTION

Syntax Form:

POS ({ string constant / string variable } , { string constant / string variable } , numeric expression)

Descriptive Form:

POS (string to be searched , substring to be found , starting location for search)

Purpose

The POS (Position) function searches for and returns the position of the first occurrence of the specified substring within the specified string. The search begins at the specified starting location and proceeds from left to right.

Explanation

The first parameter of the POS function specifies the string to be searched; the second parameter specifies the substring to be found; and the third parameter specifies the starting point for the search. The first two parameters can be specified as string constants enclosed in quotation marks or as string variables. The third parameter is specified as a numeric expression. This expression is reduced to a numeric constant and rounded to an integer when the function is evaluated. All three parameters must be enclosed in parentheses. The following example illustrates how the POS function works:

```
200 LET A$ = "HippoLionMonkeyElephantTigerGiraffe"
210 LET X = POS (A$,"Monkey",1)
```

In line 200 above, the names of six animals are assigned to the string variable A$. In line 210 a search is made for the substring "Monkey" starting at character position 1 (lefthand side). Since the substring "Monkey" starts at character position 10, the number 10 is assigned to the numeric variable X. The variable X can now be used to specify the starting position of "Monkey" in other string functions. This type of operation can be performed on a string before a substring is read with the SEG function or replaced with the REP function. The starting position of a substring must be specified in these functions, and if the starting position is unknown, it can be located with the POS function.

If the POS function is executed and the search fails to locate the specified substring, then zero (Ø) is returned as the result of the function. If the length of the specified substring is greater than the length of the specified string, then a zero (Ø) is returned as the result. And if the length of the specified substring is greater than the length of that portion of the main string to be searched (i.e., from the specified starting position to the end) then a zero (Ø) is returned.

When the search for the substring is conducted, the BASIC interpreter looks for the exact specification of the substring; upper case letters are not equal to lower case letters unless the SET CASE environmental parameter is set.

THE READ STATEMENT

Modified Syntax Form:

[Line number] REA [I/O address] string variable [, string variable]

Purpose

The READ statement assigns character strings to string variables from the DATA statement, the internal magnetic tape unit, and external peripheral devices. Incoming character strings must be in binary format.

Explanation

Reading Character Strings in the DATA Statement

The READ statement causes the BASIC interpreter to retrieve character strings from the DATA statement and assigns those strings to the specified string variables. For example:

```
41Ø READ M$
```

When this statement is executed, the DATA statement pointer must be positioned over a character string in the DATA statement. The BASIC interpreter assigns this string to M$. If the DATA statement pointer is pointing to numeric data, then an error occurs and program execution is aborted. Refer to the Input/Output Operations section for complete information on how to READ data from the DATA statement.

Reading Character Strings from a Magnetic Tape File

The READ statement can also input character strings from the current magnetic tape file. The magnetic tape read head must first be positioned at the beginning of a character string and the character string must be in binary format. Normally, the information brought into the machine with the READ statement is first stored on magnetic tape with the WRITE statement. The following is a typical READ statement specifying the internal magnetic tape unit:

brought into the machine with the READ statement is first stored on magnetic tape with the WRITE statement. The following is a typical READ statement specifying the internal magnetic tape unit:

```
560 READ @33:A$
```

This statement causes the magnetic tape unit to send the character string presently under the readhead. The BASIC interpreter assign the character string to A$. If the item read is not a character string, or if it is not stored in machine dependent binary code, then an error occurs and program execution is aborted. Refer to the Input/Output Operations section for complete information on accessing magnetic tape files with the READ statement.

Reading Character Strings from External Peripheral Devices

Reading character strings from external peripheral devices is the same as reading character strings from an internal magnetic tape file, except that the appropriate primary address must be specified. For example:

```
670 READ @8: F$,H$
```

This statement causes peripheral device number 8 on the General Purpose Interface Bus to send two character strings to the BASIC interpreter. The first string is assigned to F$, and the second string is assigned to H$. If the items are not character strings, or if they are not in binary format, then an error may occur.

Refer to the Input/Output Operations section for complete information on sending and receiving data from an external peripheral device.

THE REP FUNCTION

Syntax Form:

[Line number] [LET] string variable = REP ({ string constant / string variable } ,

numeric expression , numeric expression)

Descriptive Form:

[Line number] [LET] target string = REP (characters to be inserted , starting character position ,

number of characters to be deleted before insertion)

Purpose

The REP (Replace String) function deletes the specified number of characters from the specified target string starting at a specified character position and replaces the deleted characters with the specified substring. Once the REP function is executed, the target string is permanently changed.

Explanation

The target string is specified first and must be represented by a string variable. The line number and the keyword LET are optional. The target string contains the characters to be replaced. An equal sign and the keyword REP are specified next, then three parameters enclosed in parentheses. The first parameter specifies the characters to be inserted into the target string, the second parameter specifies the starting character position for the insertion operation, and the third parameter specifies the number of characters to be deleted in the target string before the new characters are inserted. (Characters are deleted starting at the position specified by the second parameter.)

The following example illustrates how the REP function works:

```
300 A$ = "The maid did it!"
310 A$ = REP ("butler",5,4)
320 A$ = REP ("and the maid ", 12,0)
330 A$ = REP ("",12,13)
```

In line 3ØØ the character string "The maid did it!" is assigned to the string variable A$. In line 31Ø this character string is changed; the word "maid" is replaced by the "butler." Line 31Ø tells the BASIC interpreter to fetch the string constant assigned to A$; starting at character position 5, delete 4 characters (the word "maid"), and then insert "butler" starting at character position 5. The result is:

A$ = "The butler did it!"

In line 32Ø the string constant assigned to A$ is changed again; however, this is an insertion operation only. This time the substring " and the maid " is inserted at character position 12; no characters are deleted because the third parameter is specified as Ø. The result is:

A$ = "The butler and the maid did it!"

In line 33Ø a deletion operation is performed. The thirteen characters "and the maid " are deleted from A$ and no characters are added because a null string is specified as the first parameter. The result is:

A$ = "The butler did it"

Notice that in each case, the target string A$ is permanently altered by the REP function.

THE SEG FUNCTION

Syntax Form:

[Line number] [LET] string variable = SEG ({ string constant / string variable } ,

numeric expression , numeric expression)

Descriptive Form:

[Line number] [LET] string variable = SEG (source string , starting location of substring ,

number of characters in substring)

Purpose

The SEG (Segment) function locates the specified substring within the specified string and assigns the substring to the specified string variable.

Explanation

The first parameter of the SEG function specifies the source string which contains a substring to be segmented out; the second parameter is a numeric expression which specifies the starting location of the substring; and, the third parameter is a numeric expression which specifies the number of characters in the substring. The following example illustrates how the SEG function works:

Statement	Result
100 A$ = "HippoLionMonkeyElephantTigerGiraffe"	
110 B$ = SEG (A$,10,6)	
120 PRINT B$	Monkey

In line 100 above, a character string containing the names of six animals is assigned to the string variable A$. In line 110, the SEG function is used to locate and assign the substring "Monkey" to the string variable B$. The first parameter in the SEG function specifies the source string (A$). The second parameter (10) specifies the starting location for the first character in Monkey. (Note: spaces in the source string (if any) are counted as character positions.) The third parameter specifies the number of characters in the substring; in this

case, the name Monkey contains six characters. When line 11Ø is executed, a copy of the substring Monkey is made and assigned to B$. The source string A$ is not altered or changed. Line 12Ø prints the substring assigned to B$; in this case, Monkey is printed on the GS display.

The second and third parameters of the SEG function are specified as numeric expressions. The BASIC interpreter reduces these numeric expressions to numeric constants and rounds the constants to positive integers.

The second parameter (the starting location of the substring) must be less than the number of characters in the specified source string; if not, a "null" string is returned and the program continues executing. If the second parameter is zero or negative, then the first character in the source string is considered to be the starting location of the substring.

If the third parameter (the number of characters in the substring) is a negative integer or zero, then a null or empty string is assigned to the specified string variable. If the third parameter is greater then the number of characters between the specified starting location and the end of the source string, then the SEG operation stops at the end of the source string; the characters up to that point are assigned to the specified string variable and an error message is not returned to the GS display.

The target string variable (B$ in this case) must be dimensioned large enough to accept the specified substring; if not, an error occurs and program execution is aborted.

THE STR FUNCTION

Syntax Form:

[Line number] [LET] string variable = STR (numeric expression)

Purpose

The STR (String) function converts the result of the numeric expression to a character string and assigns the character string to the specified string variable. The STR function is the inverse of the VAL function.

Explanation

The parameter of the STR function must be a valid numeric expression enclosed in parentheses. If the expression contains numeric variables, then the variables must have assigned values by the time the function is evaluated, or an error occurs. The numeric expression as a whole is reduced to a numeric constant. Once this conversion is complete, the number can no longer be used in math operations without reconversion using the VAL function.

The character string is converted using the default PRINT format. (See PRINT in the Input/Output Operations section.) The character string is assigned to the specified string variable. For example, the following statements are valid BASIC statements containing the STR function:

Statement	Result
87Ø LET A$=STR(987.6)	A$=" 987.6"
88Ø V$=STR(3+4−7)	V$=" Ø"
98Ø K$=STR(−6854ØØØØØØ)	K$=" −6.854E+9"

The string variable generated by the STR function will always have a blank (space character) as the first character position.

THE VAL FUNCTION

Syntax Form:

VAL { string constant / string variable }

Purpose

The VAL (Value) function converts a number, which is expressed as a character string, into a number which can be used as numeric data for math operations. The VAL function is the inverse of the STR function.

Explanation

The parameter of the VAL function must be a character string which contains a number in valid numeric form. When the VAL function is executed, the number in the character string is converted to numeric data. For example:

```
VAL "123.4"
  123.4
```

In this case, the VAL function is executed directly from the GS keyboard. The parameter is a number expressed as the character string "123.4". In this form, the number can not be used in math computations. When the VAL function is executed, the string "123.4" is converted to the real number 123.4. This number can now be used in numeric expressions and treated as numeric data.

If the VAL function is used in a program, then it must be part of a numeric expression or assigned to a numeric variable. For example, the following statements are valid BASIC statements containing the VAL function:

```
120 LET D5=VAL A$
130 IF VAL Z$ = .025 THEN 325
140 G=VAL P$+VAL Q$
```

Valid numeric characters are digits Ø through 9, decimal point (.),e, E, +, −, leading spaces, and trailing spaces. All other characters are ignored if they precede the first valid number; or they act as delimiters if they follow the first valid number in the string. For example:

```
25Ø LET W2 = VAL "MARK 45.2 mV / 83.4 s"
```

When this statement is executed under program control, the BASIC interpreter evaluates the character string "MARK 45.2 mV / 83.4 s" and assigns 45.2 to the numeric variable W2. The alpha characters preceding 45.2 are ignored; in this case MARK is ignored. The space following 45.2 acts as a delimiter and terminates the operation; this space and all remaining characters in the string are ignored. The number 45.2 is the result of the function.

PROGRAM EDITING

Section 11

PROGRAM EDITING

INTRODUCTION TO PROGRAM EDITING

Program Editing

The LIST, DELETE, and RENUMBER statements play an important role in program editing operations. The LIST statement causes the system to list the current program or a selected portion of the current program on the GS display or a specified peripheral device. Once the program is listed it can be examined and changed, if necessary, to meet its objectives.

Program lines can be deleted from memory with the DELETE statement and new lines added by making entries from the GS keyboard or the APPEND statement. When the editing job is complete, the entire program can be renumbered with the RENUMBER statement to make the line number increment uniform and consistent.

Sometimes it is necessary to rearrange major portions of a program. If this is the case, the job can be accomplished by exercising the SAVE and APPEND statements. A subroutine can be transferred to magnetic tape, for example, with the SAVE statement, deleted from the current program with the DELETE statement, then inserted elsewhere in the program with the APPEND statement. (Refer to the Input/Output Operations section for complete information on SAVE and APPEND.)

Program Debugging

Occasionally an error occurs while a program is running due to conditions which are set up by changing variables. To help locate the source of these hard to find run-time errors, the TRACE feature of the Graphic System BASIC language can be used. Setting TRACE causes the system to print the line number of each statement on the GS display before the statement is executed. This allows you to monitor the execution sequence of a program through branches and loops and determine the exact point where a run-time error occurs.

Program Documentation

Programs can be self-documentating by inserting REMARK statements in the appropriate spots to explain the purpose of branches, loops. and subroutines. These REMARK statements are listed and saved as part of the program, however, they are ignored by the BASIC interpreter when the program is executed.

THE DELETE STATEMENT

Syntax Form:

[Line number] DEL { ALL / variable list / line number [, line number] }

Descriptive Form:

[Line number] DELETE { ALL (entire memory) / variables to be deleted / line number [starting , line number ending] }

Purpose

The DELETE statement logically removes the specified BASIC statements or the specified variables from the read/write random access memory. If DELETE ALL is specified, an INIT command is also executed.

Explanation

Deleting Variables

If the statement DELETE ALL is executed, the BASIC interpreter clears the entire program, including defined variables, from the read/write random access memory and executes an END statement.

If variables are specified as parameters in the DELETE statement, such as...

```
DELETE A, B, C$, D5, E1
```

then the assigned values of the specified variables are cleared, and the variables enter an undefined state.

Deleting Program Statements

If a line number is specified as the parameter in a DELETE statement, such as...

```
DELETE 5ØØ
```

then the specified statement is cleared from memory; in this example, statement 5ØØ is deleted and cannot be recovered.

If two line numbers are specified as parameters in a DELETE statement, such as the statement...

```
DELETE 5ØØ, 1ØØØ
```

then all the statements between the specified statements are logically removed from memory; in this case, statements 5ØØ through 1ØØØ are removed from memory and cannot be recovered. In addition, the following statement is allowed:

```
5ØØ DELETE 4ØØ, 6ØØ
```

In this case, all statements from 4ØØ through 6ØØ (including the DELETE statement) are cleared from memory under program control.

A Word of Caution.

Once a DELETE statement is executed, the deleted information can not be recovered unless it is first stored on an external media such as magnetic tape. Refer to the Input/Output Operations section for information on storing programs and data on an external media.

NOTE

The deleted information is not actually removed from memory until the system needs additional memory space. However, the deleted information is "tagged" as such and cannot be recovered. This is what is meant by the phrase "logically removed from memory."

THE LIST STATEMENT

Syntax Form:

[Line number] LIS [I/O address] [line number [, line number]]

Descriptive Form:

[Line number] LIST [I/O address] [line number [starting , line number ending]]

Purpose

The LIST statement sends a list of the current BASIC program to the specified peripheral device. If a peripheral device is not specified, then the list is printed on the GS display.

Explanation

Listing a Program

If the LIST statement is executed without specifying line numbers as parameters, the current program is sent to the specified peripheral device as a series of ASCII character strings. If a peripheral device is not specified by entering an I/O address, then the GS display is selected as the output device by default. For example, the statement...

```
LIST
```

causes a complete listing of the current BASIC program to be printed on the GS display. The list starts with the lowest line number in memory and ends with the highest line number in memory.

Listing One Line in the Current Program.

One line in the current program can be listed by specifying the line number as a parameter in the LIST statement. For example:

```
LIST 200
```

This statement causes line number 200 to be printed on the GS display. The same function can also be executed by entering the number 200 and pressing the RECALL LINE key on the GS keyboard.

Listing a Portion of the Current Program.

A portion of the current program can be listed by specifying the starting and ending line numbers. The line numbers are specified as follows:

```
LIST 5ØØ, 75Ø
```

This statement causes the BASIC interpreter to list program lines 5ØØ through 75Ø on the GS display. If the first line number doesn't exist (5ØØ in this case) then the next highest line number in memory is listed as the first statement. If the second line number doesn't exist (75Ø in this case) then the next highest line number in memory is listed as the last statement.

Specifying an Output Device.

A list of the current program can be sent to any peripheral device in the system by specifying the appropriate primary address in the LIST statement. For example, the statement...

```
LIST @2Ø:
```

causes the BASIC interpreter to send a copy of the current program to device number 2Ø on the General Purpose Interface Bus. The entire program is converted to a series of ASCII character strings with carriage returns (CR) separating each statement. All ASCII control characters are converted to a letter—backspace—underline sequence before the list is sent to the output device. This causes some output devices to print an underlined letter which is the symbolic representation of the control character. (If the control character were sent, it might cause the output device to execute a control function.)

Specifying the primary address of the peripheral device is the only requirement for an I/O address. The BASIC interpreter automatically issues the secondary address 19 which tells the peripheral device that the ASCII strings represent a program to be listed.

The Difference between LIST and SAVE.

The statements LIST and SAVE are similar in that both statements cause the BASIC interpreter to send a copy of the current program to the specified peripheral device as ASCII character strings. The only difference is that control characters are not converted to a letter—backspace—underline sequence in the SAVE statement; the actual control characters are sent to the peripheral. In addition, the default I/O addresses are different. If an I/O address is not specified in a LIST statement, then I/O address @32,19: is issued by default. Primary address 32 selects the GS display as the output device; secondary address 19 tells the GS display to interpret the ASCII string as a program to be listed. If an I/O address is not specified in a SAVE statement, then I/O address @33,1: is issued by default. Primary address 33 selects the internal magnetic tape unit as the output device; secondary address 1 tells the magnetic tape unit that the ASCII string represents a program to be saved.

THE REMARK STATEMENT

Syntax Form:

[Line number] REM [any characters except CR]

Descriptive Form:

[Line number] REMARK [program documentation comments]

Purpose

The REMARK statement allows program documentation comments to be inserted anywhere in a program listing. This makes the program self-documenting.

Explanation

A REMARK statement can be inserted anywhere in a BASIC program to help explain what the program is doing. All REMARK statements are retained as part of the program and are output along with the other statements in memory during a LIST and SAVE operation. During program execution, however, all REMARK statements are ignored by the BASIC interpreter.

Normally, REMARK statements are inserted into a program to explain the purpose of subroutines and branches. For example, the following statement is appropriate to place at the beginning of a graphics routine which shades bar graphs:

```
340 REM This subroutine shades the blocks on a bar graph.
```

Any characters can be entered as part of the REMARK statement. The total number of characters is limited to 72 for any one statement. This includes both the keyword REM and the line number. Quotation marks are not required around the remark.

THE RENUMBER STATEMENT

Syntax Form:

[Line number] REN [numeric expression [, numeric expression [, line number]]]

Descriptive Form:

[Line number] RENUMBER [new starting line number [, increment between new line numbers [, starting line number in current program]]]

Purpose

The RENUMBER statement causes the BASIC interpreter to renumber the lines in the current program. The specified parameters provide directions for the renumber operation. If parameters are not specified, then the BASIC interpreter renumbers all program statements with line numbers greater than 10Ø. These statements are renumbered with an increment of 1Ø starting with line number 10Ø.

Explanation

The parameters of the RENUMBER statement specify which statements are to be renumbered and how they are to be renumbered. The parameters are optional and if not specified the BASIC interpreter renumbers the program according to the default parameters (10Ø, 1Ø, 10Ø).

Statement numbers specified in GOTO, GOSUB, ON...THEN..., LIST, DELETE, SAVE, APPEND, RENUMBER, RUN, IF...THEN..., RESTORE, and PRINT USING statements automatically adjusted to conform to the renumbered statements.

Specifying the New Starting Line Number.

The new starting line number is specified as the first parameter after the keyword RENUMBER. This line number is assigned to the first statement in the current program which is renumbered. For example:

```
REN 4ØØ
```

This statement causes the BASIC interpreter to assign line number 4ØØ to the first statement which is renumbered. Because this is the only parameter specified in this example, the BASIC interpreter assumes that all statements with line numbers equal to or greater than 1ØØ are to be renumbered; therefore, the first statement in the program with a line number equal to or greater than 1ØØ is renumbered as line 4ØØ.

Specifying the Increment Between Line Numbers.

The second parameter in the RENUMBER statement specifies the increment to be used for the renumbering operation. For example:

```
REN 1ØØØ, 2Ø
```

This statement causes the BASIC interpreter to renumber all statements in the current program which have line numbers equal to or greater than 1ØØ. These lines are renumbered starting at line number 1ØØØ and increase with an increment of 2Ø. If the current program contains five statements with line numbers equal to or greater than 1ØØ, for example, then the statements are renumbered 1ØØØ, 1Ø2Ø, 1Ø4Ø, 1Ø6Ø, and 1Ø8Ø. The increment between line numbers cannot be specified without first specifying a new starting line number. If the increment between line numbers is not specified, then an increment of 1Ø is used.

If the second parameter specified is a numeric expression, the BASIC interpreter evaluates and reduces the expression to a numeric constant and rounds the constant to an integer. The integer must fall within the range 1 to 65535 or an error occurs and an error message is printed on the GS display.

Specifying the Starting Line Number in the Current Program.

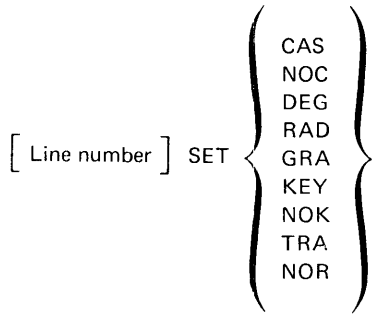
The third parameter specifies the first statement in the current program to be renumbered; all statements with line numbers equal to or greater than this line number are renumbered. The third parameter can not be entered without first specifying a new starting line number and an increment between line numbers. If the third parameter is not specified, then the BASIC interpreter renumbers all statements with line numbers equal to or greater than 1ØØ. If the specified line number doesn't exist, then the BASIC interpreter renumbers all statements with line numbers greater than the specified line number.

A Note about Sequential Order.

The renumbering process can not be used to alter the sequential order of the statements in the current program or interlace new statements with old statements. If an attempt is made to do either and the system is operating under program control, then an error occurs and program execution is aborted.

THE SET STATEMENT

Syntax Form:



Descriptive Form:

[Line number] SET environmental condition

Purpose

The SET TRACE form of the SET statement places the system in the TRACE mode of operation. The TRACE mode causes the BASIC interpreter to print the line number of a statement on the GS display before the statement is executed.

Explanation

Setting TRACE causes the BASIC interpreter to print the line number of a statement before it is executed. The line number is printed starting at the present position of the cursor; the BASIC interpreter then executes a carriage return (CR), then executes the statement. Normally, if the program does not involve graphic statements, the line numbers are printed in a single column on the left side of the GS display. When the screen is full, program execution halts until a PAGE is executed to erase the screen. After the screen is erased, program execution continues.

The TRACE feature allows you to monitor the execution order of the current program; it is a valuable aid in finding the source of run-time errors which occur as special conditions are set up during program execution. Program execution can be monitored by setting TRACE, then executing RUN; or TRACE can be set, then the program can be executed one statement at a time by pressing the STEP PROGRAM key on the GS keyboard.

To disable the TRACE feature, the statement SET NORMAL is executed from the GS keyboard or under program control. This feature is automatically set to NORMAL on system power up and when the INIT statement is executed. (Refer to the Section on Environmental Control for complete information on the SET statement.)

LANGUAGE SYNTAX

Section 12

LANGUAGE SYNTAX

INTRODUCTION

This section provides detailed information of the rules for entering BASIC statements into memory. If you have questions regarding the parameters of a keyword, or if the BASIC interpreter rejects a BASIC statement and you can't figure out why, you can find the answer here.

SYNTAX AND DESCRIPTIVE FORMS DEFINED

The term "syntax" refers to the rules governing statement structure in a programming language. The syntax governing the Graphic System BASIC language is described in syntax forms. There is one syntax form for each keyword in the language. For example:

[Line number] ROT numeric expression

This syntax form says that the keyword ROTATE can be structured into a BASIC statement starting with an optional line number, followed by the three upper case letters ROT, followed by a numeric expression.

At first glance, it's not obvious that the keyword ROT means ROTATE; and it's not at all obvious that the numeric expression is the rotation angle measured in the current trigonometric units for the system. To help clarify the meaning of the syntax forms, most syntax forms are accompanied by a descriptive form. The following form is the descriptive form for the ROTATE statement:

[Line number] ROTATE rotation angle measured in the current trigonometric units

By comparing the syntax form with the descriptive form, it can be seen that the keyword ROT means ROTATE and the numeric expression is the rotation angle measured in the current trigonometric units for the system (radians, degrees, or grads). The syntax form and the descriptive form work together to give you the most complete information on a keyword and its parameters. Remember, however, that the descriptive form is only provided to help clarify the meaning of the syntax form and should not be considered an exact description of the syntax. For a written explanation of each keyword, with examples, use the index as a guide to locate the keyword in other sections of this manual.

SYNTAX ERRORS

If the syntax of a BASIC statement is incorrect, the BASIC interpreter does not place the statement in memory until the error is corrected. This applies to statement entries from the keyboard as well as statements coming in from the internal magnetic tape or an external peripheral device.

Incorrect statements are returned to the GS display with a down arrow pointing to the approximate location of the error. The error can be corrected by using the Line Editor keys or the line buffer can be cleared completely by pressing the CLEAR key and the statement can be re-entered from scratch.

DELIMITERS USED FOR STATEMENT ENTRY

Delimiters are characters which separate the elements in a BASIC statement. The following characters are valid delimiters used in the Graphic System BASIC language.

Delimiter	Symbol
Space	blank or ♭
Comma	,
Semicolon	;
Colon	:
Quotation Mark	"

The statement below is an example of a statement containing eight delimiters. An arrow points to each delimiter.

```
200 PRINT @15,10: A$,25.6,Z$;B$
```

If a delimiter is left out or if a delimiter is incorrectly inserted into a BASIC statement, the BASIC interpreter returns the statement to the display with a down arrow pointing to the approximate location of the error. For example:

```
SYNTAX ERROR              ↓
100 PRINT "DATA ENTER ▒
```

This statement is returned to the display because the end quotation mark was left off the character string. The down arrow points to the location of the error and the cursor is placed in that location. Entering a quotation mark and pressing the RETURN key corrects the error and the statement is entered into memory.

LINE NUMBERS

When a BASIC statement is preceded by a line number, the BASIC interpreter treats the statement as a program instruction to be executed at a later time. The statement is stored in memory as part of the current BASIC program. If a line number does not precede a statement, the BASIC interpreter evaluates the statement immediately and returns the result (if any) to the GS display

Line numbers must be positive integers within the range 1 through 65535. Line numbers are optional for every keyword except FOR, DEF FN, and ON...THEN.

Except in one case, a space is not required between the line number and the keyword. Here is the special case:

```
14E1=139
```

The above entry can be interpreted two different ways. First, it can mean the entry is a program statement containing an assignment preceded by the line number 14. If this is the case, the numeric variable E1 is assigned the value 139. (The optional keyword LET is left out.)

The above entry can also be interpreted as a logical comparison. The two numbers 14E1 and 139 are compared and the logical result Ø is returned to the GS display; thus indicating the two numbers are not equal. In this unique situation, the BASIC interpreter assumes the latter operation is intended and returns a logical zero. If the statement is intended to be an assignment statement, a space is required between the line number 14 and the numeric variable E1 (i.e., 14 E1 = 139).

KEYWORDS

Keywords are alphabetic symbols which describe the function of a BASIC statement to the BASIC interpreter. The letters of a keyword can be entered as either upper or lower case. The BASIC interpreter, however, converts lower case letters to upper case for program listings. Keywords with more than three letters can be abbreviated. For example, the keyword WINDOW can be entered as WIN, WIND, WINDO, or WINDOW. The first three letters are required for an abbreviation (except for two letter keywords like OF, TO, etc.). Correct spelling is also required.

The space after each keyword is automatically generated for program listings.

Only the first two or three letters of a keyword are specified in the syntax form to show the minimum requirements for keyword entry. The keyword is spelled out completely in the descriptive form, however, to illustrate the common entry format.

OPTIONAL ENTRIES

Items enclosed in square brackets are optional. The statement is valid if these items are left out. For example:

[Line number] RUN [line number]

This statement can be entered in any one of the following forms:

```
    RUN
    RUN 83Ø
100 RUN
100 RUN 83Ø
```

OPTIONAL ENTRIES WITHIN OPTIONAL ENTRIES

Optional entries within optional entries cannot be entered by themselves. For example:

[Line number] LIS [I/O address] [line number [, line number]]

This statement can be entered in any one of the following forms:

```
LIST
LIST 1Ø
   or
LIST 1Ø,25Ø
```

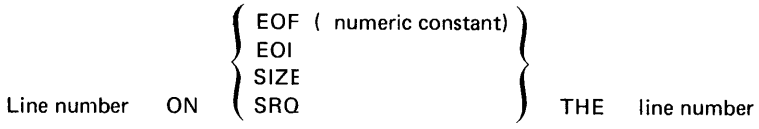
This statement cannot be entered in as follows:

```
LIST ,25Ø
```

In this example, the second line number within the brackets (and its accompanying comma) cannot be entered without first entering the first line number in the brackets.

IT'S A MATTER OF CHOICE

Items enclosed in braces make up a selection list from which one item must be selected. For example,



This statement can be entered in one of the following forms:

Line number	ON	EOF	(numeric expression) THEN line number
Line number	ON	EOI	THEN line number
Line number	ON	SIZE	THEN line number
Line number	ON	SRQ	THEN line number

I/O ADDRESS

The term "I/O address" in a syntax form is a substitute for the following syntactical description:

Syntax Form:

{%} {@} numeric expression [, numeric expression] :

Descriptive Form:

{%} {@} primary address [, secondary address] :

I/O addresses are specified in a statement by entering either an "at" sign (@) or a percent sign (%) followed by a primary address, followed by an optional secondary address, followed by a colon (:). The "at" sign (@) and the percent sign (%) specify the type of delimiters used in the transfer, the primary address selects the input source or the output destination - whichever is appropriate, the secondary address tells the peripheral device what function is being performed by the BASIC interpreter, and the colon (:) is used as the I/O address delimiter.

Specifying Delimiters to be Used in the Transfer. If the "at" sign is specified at the beginning of the I/O address, standard delimiters are used during the I/O operation. The standard delimiters are CR (Carriage Return) for record separator and hexidecimal FF for the End Of File mark unless the standard is changed with an environmental setting. (Refer to the topic Processor Status in the Environmental Control section for details.)

If the % sign is specified instead of the "@" sign, an alternate record separator and End Of File character is used on INPUT operations. The alternate delimiters are specified in an environmental setting. (Refer to the topic Processor Status in the Environmental Control section for details.)

Primary address. In every I/O statement except WBYTE and RBYTE, the primary address is specified as a peripheral device number. Peripheral devices are assigned peripheral device numbers via hardware connections based on the following guidelines:

Device Number	Peripheral Device
1-3Ø	External peripheral devices on the General Purpose Interface Bus
31-39	Internal peripheral devices connected directly to the microprocessor bus lines
4Ø-255	Reserved for future use

Internal peripheral devices are preassigned the following peripheral device numbers:

Device Number	Peripheral Device
31	GS keyboard
32	GS display
33	Magnetic Tape Unit
34	DATA Statement
35	Unassigned
36	Unassigned
37	Processor Status
38	Unassigned
39	Unassigned

Peripheral device numbers can be specified as a numeric expression in a statement as long as the BASIC interpreter can reduce the expression to a numeric constant and round the constant to an integer within the range Ø to 255. This means that the primary address can be specified as a numeric variable; by changing the value assigned to the variable, different peripheral devices can be selected as the input source or output destination for a particular I/O statement.

Secondary Addresses. Secondary addresses are positive integers within the range Ø to 32. Like primary addresses, secondary addresses can be specified as numeric expressions as long as the BASIC interpreter can reduce the entry to a numeric constant and round the constant to an integer within the range Ø to 32. (If 32 is specified as a secondary address, the BASIC interpreter is inhibited from issuing a secondary address.)

Default Primary and Secondary Addresses. If an I/O address is not specified, the BASIC interpreter issues a default primary and secondary address for the keywords shown in the following diagram. For example, if a LIST statement is executed without specifying an I/O address, the BASIC interpreter issues the primary address 32 which tells the GS display it is the destination target for the upcoming data transfer. The BASIC interpreter then issues the secondary address 19 which tells the display a LIST operation is about to take place.

If a primary address is specified in the LIST statement which outputs the list to a device on the General Purpose Interface Bus, for example, and if a secondary address is not specified, the BASIC interpreter issues 19 as a default secondary address. The default secondary address 19 may or may not have meaning to the I/O device.

DEFAULT I/O ADDRESSES	
APPEND	@ 33,4:
BRIGHTNESS	@ 32,30:
CHARSIZE	@ 32,17:
CLOSE	@ 33,2:
COPY	@ 32,10:
DASH	@ 32,31:
DRAW	@ 32,20:
FIND	@ 33,27:
FONT	@ 32,18:
GIN	@ 32,24:
HOME	@ 32,23:
INPUT	@ 31,13:
KILL	@ 33,7:
LIST	@ 32,19:
MARK	@ 33,28:
MOVE	@ 32,21:
OLD	@ 33,4:
PAGE	@ 32,22:
PRINT	@ 32,12:
RDRAW	@ 32,20:
READ	@ 34,14:
RMOVE	@ 32,21:
SAVE	@ 33,1:
SECRET	@ 37,29:
TLIST	@ 32,19:
WRITE	@ 33,15:

For a description of the meaning of each secondary address, refer to Appendix B.

DATA ITEMS

If a data item is specified in the syntax form, either a numeric constant or a string constant can be entered.

VARIABLE LIST

If a variable list is specified, any number of numeric variables or string variables or combinations thereof can be entered. The variables must be separated by commas, unless otherwise specified.

LINE NUMBER LIST

If a line number list is specified, any number of line numbers can be entered. The line numbers must be separated by commas, unless otherwise specified.

TARGET VARIABLES

If a variable is described as a target variable in the descriptive form, then the variable receives the numeric or string constant which results when the statement is executed. Usually, incoming data items from an I/O device are assigned to target variables. String functions and array operations must always have target variables.

TRAILING DOTS ...

If a syntax form ends in trailing dots, the preceding element can be repeated as many times as desired. For example, the syntax form ...

$$\left[\text{Line number}\right] \text{ DAT } \left\{\begin{matrix}\text{string constant}\\ \text{numeric constant}\end{matrix}\right\} \left[, \left\{\begin{matrix}\text{string constant}\\ \text{numeric constant}\end{matrix}\right\}\right] \ \ldots$$

means the element $\left[, \left\{\begin{matrix}\text{string constant}\\ \text{numeric constant}\end{matrix}\right\}\right]$ can be repeated as many times as desired.

SUBSTITUTING ELEMENTS

Numeric Expressions. If a numeric expression is specified in the syntax form, you can enter a numeric expression, a numeric function, a numeric variable, a subscripted array variable, a logical or relational comparison enclosed in parenthesis, or a numeric constant. The system must, however, be able to reduce the entry to a numeric constant.

Array Variables. If an array variable is specified, only an array variable previously dimensioned by the DIM (Dimension) statement can be entered.

Numeric Variables. If a numeric variable is specified, a subscripted array variable is also allowed, except in the FOR, NEXT, DEF FN, and POLL statements.

Numeric Constant. If a numeric constant is specified, only a numeric constant can be entered.

String Constant. If a string constant is specified, only a string constant can be entered.

String Variable. If a string variable is specified, only a string variable can be entered.

PARENTHESES AROUND THE PARAMETERS OF FUNCTIONS

If the parameter of a function is a numeric expression, the expression must be enclosed in parentheses, as shown in the syntax form; otherwise, the parentheses are optional. For example, if the BASIC interpreter evaluates the statement LET Y = SIN 3+4, the BASIC interpreter assumes 3 is the function's parameter and takes the sine of 3, adds 4, and assigns the result to the numeric variable Y. If parentheses are not used, as shown, the BASIC interpreter assumes the first element following a function is the parameter. If, however, the entire expression 3+4 is the parameter of the function, then the expression must be enclosed in parentheses as in the statement LET Y = SIN (3+4). When this statement is evaluated, the BASIC interpreter first adds 3 and 4 to get 7, takes the sine of 7, and assigns the result to the numeric variable Y. When listing a program, the BASIC interpreter always places parentheses around the parameter of a function to make the listing easier to read.

KEYWORDS WITH SYNTAX AND DESCRIPTIVE FORMS

The following is a complete alphabetical list of all the keywords in the Graphic System BASIC language with their syntax and descriptive forms.

THE ABS FUNCTION

Syntax Form:

ABS numeric expression

THE ACS FUNCTION

Syntax Form:

{ ACOS / ACS } numeric expression

THE APPEND STATEMENT

Syntax Form:

[Line number] APP [I/O address] line number [, numeric expression]

Descriptive Form:

[Line number] APPEND [I/O address] target line number in current program

[, increment between line numbers]

THE ASC FUNCTION

Syntax Form:

ASC { string constant / string variable }

THE ASN FUNCTION

Syntax Form:

{ ASIN / ASN } numeric expression

THE ATN FUNCTION

Syntax Form:

{ ATAN / ATN } numeric expression

THE AXIS STATEMENT

Syntax Form:

[Line number] AXI [I/O address] [numeric expression , numeric expression
[, numeric expression , numeric expression]]

Descriptive Form:

[Line number] AXIS [I/O address] [X axis tic interval in user data units ,
Y axis tic interval in user data units [, X axis intercept in user data units ,
Y axis intercept in user data units]]

THE BAPPEN ROUTINE

Syntax Form:

[Line number] CALL { "BAPPEN," / string variable, } [I/O address;] line number [, increment]

Descriptive Form:

[Line number] CALL routine name, [I/O address;] target line number in current program [line number , increment]

THE BOLD ROUTINE

Syntax Form:

[Line number] CALL { "BOLD" / string variable } [, I/O address]

Descriptive Form:

[Line number] CALL routine name [, I/O address]

THE BRIGHTNESS STATEMENT

Syntax Form:

[Line number] BRI numeric expression

Discriptive Form:

[Line number] BRIGHTNESS display code

THE BSAVE ROUTINE

Syntax Form:

[Line number] CALL { "BSAVE" / string variable } [, I/O address]

Descriptive Form:

[Line number] CALL routine name [, I/O address]

THE CALL STATEMENT

Syntax Form:

[Line number] CAL { string variable / string constant } [{ ; / , } { string constant / string variable / numeric expression }] ...

Descriptive Form:

[Line number] CALL routine name [{ ; / , } data item to be passed to firmware routine] ...

THE CHARSIZE STATEMENT

Syntax Form:

[Line number] CHA numeric expression

Descriptive Form:

[Line number] CHARSIZE size code

THE CHR FUNCTION

Syntax Form:

[Line number] [LET] string variable = CHR (numeric expression)

THE CLOSE STATEMENT

Syntax Form:

[Line number] CLO

Descriptive Form:

[Line number] CLOSE

THE COPY STATEMENT

Syntax Form:

[Line number] COP

Descriptive Form:

[Line number] COPY

THE COS FUNCTION

Syntax Form:

COS numeric expression

THE DASH STATEMENT

Syntax Form:

[Line number] DAS numeric expression

Descriptive Form:

[Line number] DASH dash pattern

THE DATA STATEMENT

Syntax Form:

[Line number] DAT { string constant / numeric constant } [, { string constant / numeric constant }] ...

Descriptive Form:

[Line number] DATA data item [, data item] ...

THE DEF FN STATEMENT

Syntax Form:

Line number DEF FN letter (numeric variable) = numeric expression

Descriptive Form:

Line number DEF FN any letter (numeric variable) = function to be defined

THE DET FUNCTION

Syntax Form:

[Line number] array variable – INV array variable
[[Line number] numeric variable –] DET

THE DELETE STATEMENT

Syntax Form:

[Line number] DEL { ALL | variable list | line number [, line number] }

Descriptive Form:

[Line number] DELETE { ALL (entire memory) | variables to be deleted | line number [starting , line number ending] }

THE DIM STATEMENT

Syntax Form:

[Line number] DIM { string variable (numeric expression) | numeric variable (numeric expression [, numeric expression]) }

[, { string variable (numeric expression) | numeric variable (numeric expression) [, numeric expression]) }] ...

Descriptive Form:

[Line number] DIM { string variable (maximum number of characters) | array variable (first dimension [, second dimension]) }

[, { string variable (maximum number of characters) | array variable (first dimension [, second dimension]) }] ...

THE DRAW STATEMENT

Syntax Form:

[Line number] DRA [I/O address] numeric expression , numeric expression

Descriptive Form:

[Line number] DRAW [I/O address] X coordinate in user data units , Y coordinate in user data units

THE END STATEMENT

Syntax Form:

[Line number] END

Descriptive Form:

[Line number] END

THE EXP FUNCTION

Syntax Form:

EXP numeric expression

THE FIND STATEMENT

Syntax Form:

[Line number] FIN [I/O address] numeric expression

Descriptive Form:

[Line number] FIND [I/O address] tape file number

THE FONT STATEMENT

Syntax Form:

[Line number] FON numeric expression

Descriptive Form:

[Line number] FONT font code

THE FOR STATEMENT

Syntax Form:

Line number FOR numeric variable = numeric expression TO numeric expression [STE numeric expression]

Descriptive Form:

Line number FOR index = starting value TO ending value [STEP increment for each loop]

THE FUZZ STATEMENT

Syntax Form:

[Line number] FUZ numeric expression [, numeric expression]

Descriptive Form:

[Line number] FUZZ number of digits for comparisons not involving zero [, numeric value of closeness for comparisons with zero]

THE GIN STATEMENT

Syntax Form:

[Line number] GIN [I/O address] numeric variable , numeric variable

Descriptive Form:

[Line number] GIN [I/O address] target variable for X coordinate

, target variable for Y coordinate

THE GOSUB STATEMENT

Syntax Form:

[Line number] GOS { line number / numeric expression OF line number [, line number] ... }

Descriptive Form:

[Line number] GOSUB { line number / line number selector OF line number list }

THE GO TO STATEMENT

Syntax Form:

[Line number] { GO TO / GOTO } { line number / numeric expression OF line number [, line number] ... }

Descriptive Form:

[Line number] { GO TO / GOTO } { line number / line number selector OF line number list }

THE HOME STATEMENT

Syntax Form:

[Line number] HOM [I/O address]

Descriptive Form:

[Line number] HOME [I/O address]

THE IDN ROUTINE

Syntax Form:

[Line number] CALL { "IDN" / string variable }, array variable

Descriptive Form:

[Line number] CALL routine call name , target variable

THE IF... THEN... STATEMENT

Syntax Form:

[Line number] IF numeric expression THE line number

Descriptive Form:

[Line number] IF numeric expression THEN line number

THE IMAGE STATEMENT

Syntax Form:

[Line number] IMA any characters except CR

Descriptive Form:

[Line number] IMAGE format string for the print using statement

THE INIT STATEMENT

Syntax Form:

[Line number] INI

Descriptive Form:

[Line number] INIT

THE INPUT STATEMENT

Syntax Form:

[Line number] INP [I/O address] { array variable / string variable / numeric variable } [, { array variable / string variable / numeric variable }] ...

Descriptive Form:

[Line number] INPUT [I/O address] target variables for incoming data items which are formatted in ASCII code

THE INT FUNCTION

Syntax Form:

INT numeric expression

THE INV FUNCTION

Syntax Form:

[Line number] array variable = INV array variable

Descriptive Form:

[Line number] target variable = INV parameter variable

THE KILL STATEMENT

Syntax Form:

[Line number] KIL [I/O address] numeric expression

Descriptive Form:

[Line number] KILL [I/O address] tape file number

THE LEN FUNCTION

Syntax Form:

LEN { string constant / string variable }

THE LET STATEMENT

Syntax Form:

[Line number] [LET] { array variable = numeric expression / string variable = string expression / numeric variable = numeric expression }

Descriptive Form:

[Line number] [LET] { array variable = numeric expression / string variable = string expression / numeric variable = numeric expression }

THE LGT FUNCTION

Syntax Form:

LGT numeric expression

THE LINK ROUTINE

Syntax Form:

[Line number] CALL { "LINK," / string variable, } [I/O address;] line number

Descriptive Form:

[Line number] CALL routine name, [I/O address;] line number of entry point

THE LIST STATEMENT

Syntax Form:

[Line number] LIS [I/O address] [line number [, line number]]

Descriptive Form:

[Line number] LIST [I/O address] [line number [starting , line number ending]]

THE LOG FUNCTION

Syntax Form:

LOG numeric expression

THE MARK STATEMENT

Syntax Form:

[Line number] MAR [I/O address] numeric expression , numeric expression

Descriptive Form:

[Line number] MARK [I/O address] number of files , number of bytes per file

THE MEMORY FUNCTION

Syntax Form:

MEM

THE MOVE STATEMENT

Syntax Form:

[Line number] MOV [I/O address] numeric expression , numeric expression

Descriptive Form:

[Line number] MOVE [I/O address] X coordinate in user data units , Y coordinate in user data units

THE MPY FUNCTION

Syntax Form:

[Line number] array variable = array variable MPY array variable

Descriptive Form:

[Line number] target variable = parameter variable MPY parameter variable

THE MTPACK ROUTINE

Syntax Form:

[Line number] CALL { "MTPACK" / string variable }

Descriptive Form:

[Line number] CALL routine name

THE NEXT STATEMENT

Syntax Form:

Line number NEX numeric variable

Descriptive Form:

Line number NEXT index

THE OFF STATEMENT

Syntax Form:

[Line number] OFF [{ EOF (numeric constant) / EOI / SIZE / SRQ }]

Descriptive Form:

[Line number] OFF [interrupt condition]

THE OLD STATEMENT

Syntax Form:

[Line number] OLD [I/O address]

Descriptive Form:

[Line number] OLD [I/O address]

THE ON... THEN... STATEMENT

Syntax Form:

Line number ON { EOF (numeric constant) / EOI / SIZE / SRQ } THEN line number

Descriptive Form:

Line number ON interrupt condition THEN line number

THE PAGE STATEMENT

Syntax Form:

[Line number] PAG [I/O address]

Descriptive Form:

[Line number] PAGE [I/O address]

THE PI FUNCTION

Syntax Form:

PI

THE POINTER STATEMENT

Syntax Form:

[Line number] POI numeric variable , numeric variable , string variable

Descriptive Form:

[Line number] POINTER target variable for X coordinate of graphic point in user data
units , target variable for Y coordinate of graphic point in user data
units , target variable to record the key which is pressed to end the entry

THE POLL STATEMENT

Syntax Form:

[Line number] POL numeric variable , numeric variable ; primary address [, secondary address] [; primary address [, secondary address]] ...

Descriptive Form:

[Line number] POLL target variable for device identifier , target variable for return status information ; address list

THE POS FUNCTION

Syntax Form:

POS ({ string constant / string variable } , { string constant / string variable } , numeric expression)

Descriptive Form:

POS (string to be searched , substring to be found , starting location for search)

THE PRINT STATEMENT

Syntax Form:

[Line number] PRI [I/O address] [USI { string constant / string variable / line number } :]

[{ string constant / string variable / numeric expression } [{ , / ; } string constant / string variable / numeric expression]] ... [;]

Descriptive Form:

[Line number] PRINT [I/O address] [USING { format string / format string variable / IMAGE line number } :]

[item to be printed [{ , / ; } item to be printed]] ... [;]

THE RBYTE STATEMENT

Syntax Form:

[Line number] RBY numeric variable [, numeric variable] ...

Descriptive Form:

[Line number] RBYTE target variable for incoming data byte [, target variable for incoming data byte] ...

THE RDRAW STATEMENT

Syntax Form:

[Line number] RDR [I/O address] numeric expression , numeric expression

Descriptive Form:

[Line number] RDRAW [I/O address] X increment in user data units , Y increment in user data units

THE READ STATEMENT

Syntax Form:

[Line number] REA [I/O address] { array variable / string variable / numeric variable } [, { array variable / string variable / numeric variable }] ...

Descriptive Form:

[Line number] READ [I/O address] target variables for incoming data

items formatted in machine dependent binary code

THE REMARK STATEMENT

Syntax Form:

[Line number] REM [any characters except CR]

Descriptive Form:

[Line number] REMARK [program documentation comments]

THE RENUMBER STATEMENT

Syntax Form:

[Line number] REN [numeric expression [, numeric expression [, line number]]]

Descriptive Form:

[Line number] RENUMBER [new starting line number [, increment between new line numbers [, starting line number in current program]]]

THE REP FUNCTION

Syntax Form:

[Line number] [LET] string variable = REP ({ string constant / string variable } , numeric expression , numeric expression)

Descriptive Form:

[Line number] [LET] target string = REP (characters to be inserted , starting character position , number of characters to be deleted before insertion)

THE RESTORE STATEMENT

Syntax Form:

[Line number] RES [line number]

Descriptive Form:

[Line number] RESTORE [line number]

THE RETURN STATEMENT

Syntax Form:

[Line number] RET

Descriptive Form:

[Line number] RETURN

THE RMOVE STATEMENT

Syntax Form:

[Line number] RMO [I/O address] numeric expression , numeric expression

Descriptive Form:

[Line number] RMOVE [I/O address] X increment in user data units , Y increment in user data units

THE RND FUNCTION

Syntax Form:

RND numeric expression

THE ROTATE STATEMENT

Syntax Form:

[Line number] ROT numeric expression

Descriptive Form:

[Line number] ROTATE rotation angle measured in the current trigonometric units

THE RUN STATEMENT

Syntax Form:

[Line number] RUN [line number]

Descriptive Form:

[Line number] RUN [starting line number]

THE SAVE STATEMENT

Syntax Form:

[Line number] SAV [I/O address] [line number [, line number]]

Descriptive Form:

[Line number] SAVE [I/O address] [line number [starting , line number ending]]

THE SCALE STATEMENT

Syntax Form:

[Line number] SCA numeric expression , numeric expression

Descriptive Form:

[Line number] SCALE horizontal scale factor , vertical scale factor

THE SECRET STATEMENT

Syntax Form:

[Line number] SEC [I/O address]

Descriptive Form:

[Line number] SECRET [I/O address]

THE SEG FUNCTION

Syntax Form:

[Line number] [LET] string variable = SEG ({ string constant / string variable } ,

numeric expression , numeric expression)

Descriptive Form:

[Line number] [LET] string variable = SEG (source string , starting location of substring ,

number of characters in substring)

THE SET STATEMENT

Syntax Form:

[Line number] SET { CAS | NOC | DEG | RAD | GRA | KEY | NOK | TRA | NOR }

Descriptive Form:

[Line number] SET environmental condition

THE SGN FUNCTION

Syntax Form:

{ SIGN | SGN } numeric expression

THE SIN FUNCTION

Syntax Form:

SIN numeric expression

THE SPACE FUNCTION

Syntax Form:

SPA

THE SQR FUNCTION

Syntax Form:

SQR numeric expression

THE STOP STATEMENT

Syntax Form:

[Line number] STO

Descriptive Form:

[Line number] STOP

THE STR FUNCTION

Syntax Form:

[Line number] [LET] string variable = STR (numeric expression)

THE SUM FUNCTION

Syntax Form:

SUM array variable

THE TAN FUNCTION

Syntax Form:

TAN numeric expression

THE TLIST STATEMENT

Syntax Form:

[Line number] TLI [I/O address]

Descriptive Form:

[Line number] TLIST [I/O address]

THE TRN FUNCTION

Syntax Form:

[Line number] array variable = TRN array variable

Descriptive Form:

[Line number] target variable = TRN parameter variable

THE TYP FUNCTION

Syntax Form:

TYP numeric expression

Descriptive Form:

TYP logical unit number

THE VAL FUNCTION

Syntax Form:

VAL { string constant / string variable }

THE VIEWPORT STATEMENT

Syntax Form:

[Line number] VIE numeric expression , numeric expression , numeric expression

, numeric expression

Descriptive Form:

[Line number] VIEWPORT minimum horizontal value in GDU s , maximum

horizontal value in GDU s , minimum vertical value in GDU s ,

maximum verticai value in GDU s

THE WAIT ROUTINE

Syntax Form:

[Line number] CALL { "WAIT" / string variable } [, numeric variable]

Descriptive Form:

[Line number] CALL routine name [, number of seconds]

THE WAIT STATEMENT

Syntax Form:

[Line number] WAI

Descriptive Form:

[Line number] WAIT

THE WBYTE STATEMENT

Syntax Form:

[Line number] WBY [@ numeric expression [, numeric expression] ... :]

[numeric expression] [,numeric expression] ...

Descriptive Form:

[Line number] WBYTE [@ absolute address [, absolute address] ... :]

[data bytes to be sent out over the General Purpose Interface Bus]

THE WINDOW STATEMENT

Syntax Form:

[Line number] WIN numeric expression , numeric expression , numeric expression

, numeric expression

Descriptive Form:

[Line number] WINDOW minimum horizontal (X) value in user data units , maximum horizontal (X) value in user data units , minimum vertical (Y) value in user data units , maximum vertical (Y) value in user data units

THE WRITE STATEMENT

Syntax Form:

[Line number] WRI [I/O address] { string constant / string variable / numeric expression }

[[, { string constant / string variable / numeric expression }]] ...

Descriptive Form:

[Line number] WRITE [I/O address] data item to be written in machine dependent binary code [, data item to be written in machine dependent binary code] ...

Appendix A

ERROR MESSAGES

MESSAGE NUMBER	ERROR MESSAGE
* 1	An arithmetic operation has resulted in an out of range number. Example: 1/1.ØE-3Ø8
* 2	A divide by zero operation has resulted in an out of range number. Example: 4/Ø
* 3	An exponentiation operation has resulted in an out of range number. Example: 5↑1.ØE+3ØØ
* 4	An exponentiation operation involving the base e has resulted in an out of range number. Example: EXP (1.ØE+234)
* 5	The parameter of a trigonometric function is too large. That is, the variable N in the statement A=SIN(N*2*PI) is greater than 65536. Example: A=SIN(4.2E+5) when the trigonometric units are set to RADIANS.
* 6	An attempt has been made to take the square root of a negative number. The positive square root is returned by default. Example: SQR (–4)
7	The line number in the program line is not an integer within the range 1 to 65535. Example: Ø REM THIS IS AN INVALID LINE NUMBER

**NOTE: This error is caused by a math operation which produces a predefined out of range number. This error condition can be handled by the BASIC program without terminating program execution. Refer to the ON. . .THEN. . . statement for details.*

MESSAGE NUMBER	ERROR MESSAGE
8	The matrix arrays are not conformable in the current math operation. That is, they are not of the same dimension and/or do not have the same number of elements. Example: INIT DIM A(2),B(2),C(3) A=1 B=2 C=A+B
9	A previously defined numeric variable can not be dimensioned as an array variable without deleting the numeric variable first. Example: INIT B=3 DIM B(2,2)
10	There is an error in the subscript of a variable due to one of the following reasons: 1. A numeric variable can't be subscripted. 2. A subscript is out of range. Example 1: INIT DIM A(2,2) A(2,3)=5 Example 2: INIT B=3 PRINT B(4)
11	An attempt has been made to use an undefined DEF FN function.
12	There is a parameter error in the CALL statement to a ROM pack.
13	A WBYTE parameter is not within the range −255 through +255. Example: WBYTE 3ØØ
14	A parameter for the APPEND statement is invalid.
15	An Attempt has been made to APPEND to a non-existent line number.
16	There is an invalid parameter in the FUZZ statement. Example: FUZZ −1Ø

MESSAGE NUMBER	ERROR MESSAGE
17	There is an invalid parameter in a RENUMBER operation due to one of the following reasons: 1. The first or third parameter is not a line number within the range 1 through 65535. 2. The increment (second parameter) is not within the range 1 through 65535 or is so large that out of range line numbers are generated during the RENUMBER operation. 3. Statement replacement or statement interlacing will occur if the RENUMBER operation is attempted. This error may occur during an APPEND operation.
18	An argument is out of range — domain error.
19	There is an invalid parameter in a GOTO, FOR, or NEXT statement. Example: 5ØØ FOR I=1 to 2Ø where I has been previously defined as an array variable.
20	The logical unit number specified in the statement is not within the range Ø through 9. Example: 1ØØ ON EOF (1Ø) THEN 5ØØ
21	The assignment statement is invalid because of one of the following reasons: 1. An attempt has been made to assign an array to a numeric variable. 2. Two arrays in the statement are not conformable (not of the same dimension and/or do not have the same number of elements). 3. An attempt has been made to assign a character string to a string variable and the character string is larger than the dimensioned size of the variable.
22	There is an error in an exponentiation operation because the base is less than Ø and the exponent is not an integer less than 256. Example: −1Ø↑257.5
23	An attempt has been made to take the LOG or LGT of a number which is equal to or less than Ø. Example: LOG (−1)

MESSAGE NUMBER	ERROR MESSAGE
24	The parameter of the ASN function or the ACS function is not within the range −1 to +1. Example: ASN (2)
25	The parameter of the CHR function is not within the range Ø through 127. Example: A$=CHR(128)
26	Not used.
27	The parameter is out of the domain of the function. Example: A$=STR(X) where X has been previously defined as an array variable.
28	A REP function parameter is invalid.
29	The parameter in the VAL function is not a character string containing a valid number. Example: A=VAL("Hi")
30	**The matrix multiplication operation failed because the arrays are not conformable.**
*31	The matrix inversion failed because the determinant was Ø. This error is treated as a SIZE error.
32	The routine name specified in the CALL statement can not be found. Example: CALL "FIX IT" where the routine "FIX IT" resides in a ROM pack which is not plugged into the system.
33	Not used.
34	The DATA statement is invalid because of one of the following reasons: 1. There isn't a DATA statement in the current BASIC program. 2. There is not enough data in the DATA statement from the present position of the pointer to the end of the statement. 3. An attempt has been made to RESTORE the data statement pointer to a nonexistent DATA statement.

**NOTE: This error is caused by a math operation which produces a predefined out of range number. This error condition can be handled by the BASIC program without terminating program execution. Refer to the ON. . .THEN. . . statement for details.*

MESSAGE NUMBER	ERROR MESSAGE
35	The statements DEF FN, FOR, and ON. . .THEN. . . can not be entered without a line number.
36	There is an undefined variable in the specified line. A numeric variable has not been assigned a value or an array element has not been assigned a value. Example: INIT DIM A(2,2) A(1,2) = 4 PRINT A
37	An extended function ROM (Read Only Memory) is required to perform this operation.
38	This output operation cannot be executed because the current BASIC program is marked SECRET.
39	This operation can not be executed because the Random Access Memory is full. Some program lines or variables must be deleted.
40	Not used.
41	A parity error has occurred in RAM. Although the error is nonfatal (and the message will not be repeated), further operations are unreliable until power has been turned off and back on.
42	A PAGE FULL interrupt condition has occurred.
43	A peripheral device on the General Purpose Interface Bus is requesting service and an ON SRQ THEN . . . statement has not been executed in the current BASIC program.
44	The EOI signal line on the General Purpose Interface Bus has been activated and an ON EOI THEN . . . statement has not been activated in the current BASIC program.
45	A ROM pack is requesting service and the ON UNIT for external interrupt number 1 has not been activated in the current BASIC program.
46	A ROM pack is requesting service and the ON UNIT for external interrupt number 2 has not been activated in the current BASIC program.
47	A ROM pack is requesting service and the ON UNIT for external interrupt number 3 has not been activated in the current BASIC program.
48	The end of the current file has been reached on an I/O device and an ON EOF THEN statement has not been executed in the current BASIC program.

MESSAGE NUMBER	ERROR MESSAGE
49	The statement with the specified line number is too long. This error situation occurs if an attempt is made to LIST or SAVE a BASIC program which contains a line with more than 72 characters. Sometimes a RENUMBER operation can make a line longer than 72 characters.
50	The incoming BASIC program contains a line with more than 72 characters.
51	The line number specified in this statement cannot be found or is invalid. Example: GO TO 5ØØ where the line 5ØØ doesn't exist or PRINT USING 1ØØ: where line 1ØØ isn't an IMAGE statement.
52	Either the specified magnetic tape file doesn't exist or an attempt has just been made to KILL the LAST (dummy) file.
53	After 1Ø attempts, the internal magnetic tape unit has been unable to read a portion of the current magnetic tape.
54	The end of the magnetic tape medium has been detected. Marking a file longer than the remaining portion of the tape can cause this error.
55	An attempt has been made to incorrectly access a magnetic tape file. Example: Executing a READ statement when the tape head is positioned in the middle of an ASCII data file.
56	An attempt has been made to send information to a write-protected tape. Remove the tape cartridge, rotate the write-protect cylinder until the black arrow points away from SAFE, insert the tape cartridge, and try the operation again.
57	An attempt has been made to read to or write to a non-existent tape cartridge. Insert a tape cartridge into the tape slot and try the operation again.
58	An attempt has been made to read data which is stored in an invalid magnetic tape format. The tape format must be compatible with the Graphic System standard.
59	A program was not found when the OLD statement was executed.
60	Not used.

MESSAGE NUMBER	ERROR MESSAGE
61	An attempt has been made to execute an invalid operation on an open magnetic tape file. Example: Executing a MARK statement with the tape head positioned in the middle of an open data file.
62	There is a disc file system parameter error.
63	There is an error in a binary data header, most likely caused by a machine malfunction.
64	The character string is too long to output in binary format. The length is limited to 8192 characters.
65	Not used.
66	The primary address in the specified line is not within the range 1 through 255.
67	An attempt has been made to execute an illegal I/O operation on an internal peripheral device. Example: DRAW @33:5Ø,5Ø
68	The diagnostic loader failed.
69	An input error or an output error has occurred on the General Purpose Interface Bus. Both the NDAC and NRFD signal lines are inactive high, which is an illegal GPIB state. This usually means that there are no peripheral devices connected to the GPIB.
70	There is an incomplete literal string specification in the format string. Example: 1ØØ IMAGE 6D,5("MARK
71	A format string is not specified for the PRINT USING operation.
72	Format string too short. Not enough matching data specified. Example: 1ØØ IMAGE 6D 11Ø PRINT USING 1ØØ: 23,24,25 Line 1ØØ should be: 1ØØ IMAGE 3(6D)
73	There is an invalid character in the format string specified in the PRINT USING statement.

MESSAGE NUMBER	ERROR MESSAGE
74	An n modifier in the format string is out of range or is incorrectly used. n modifiers must be positive integers within the range 1 through 11 when used with E field operator and must be within the range 1 through 255 when used with A,D,L,P,T,X,'',(, and / field operators.
75	The format string specified in the PRINT USING statement is too long (i.e., there are too many data specifiers for the PRINT statement). Example: 1ØØ IMAGE 3(6D) 11Ø PRINT USING 1ØØ:A,B Line 1ØØ should be: 1ØØ IMAGE 2(6D)
76	Parentheses are incorrectly used in the format string which is specified in the PRINT USING statement. Example: 1ØØ IMAGE 2(6D) 11Ø PRINT USING 1ØØ:A,B Line 1ØØ should be 1ØØ IMAGE 2(6D)
77	There is an invalid modifier to a field operator in the format string which is specified in the PRINT USING statement. Example: 1ØØ IMAGE 2(6D),2S 11Ø PRINT USING 1ØØ:A,B Line 1ØØ should be: 1ØØ IMAGE 2(6D),S An n modifier is not allowed
78	An S modifier is incorrectly positioned in the format string which is specified in the PRINT USING statement. The S modifier must always be positioned at the end of the format string. Example: 1ØØ IMAGE 4D,S,8A Line 1ØØ should be: 1ØØ IMAGE 4D,8A,S
79	A comma is incorrectly used in the format string which is specified in the PRINT USING statement. Example: 1ØØ IMAGE 6,D,S Line 1ØØ should be: 1ØØ IMAGE 6D,S

MESSAGE NUMBER	ERROR MESSAGE
80	A decimal point is incorrectly used in the format string which is specified in the PRINT USING statement. Example: 1ØØ IMAGE .3D 11Ø PRINT USING 1ØØ:812.345 Line 1ØØ should be: 1ØØ IMAGE FD.3D
81	A data type mismatch has occurred in the PRINT USING statement. Example: 1ØØ IMAGE 6D,6A 11Ø PRINT USING 1ØØ: "MARY",26 Line 1ØØ should be: 1ØØ IMAGE 6A,6D
82	A tabbing error has occurred in the format string which is specified in the PRINT USING statement. Example: 1ØØ IMAGE 1ØA,2T,FD 11Ø PRINT USING 1ØØ: "ENTER DATA",D The absolute tab to position 2 specified by 2T in line 1ØØ cannot occur because the cursor has already advanced beyond position 2. The tab specification must be at least 11T in this case.
83	A number specified in the PRINT USING statement contains an exponent outside the range ± 127. Example: 1ØØ IMAGE FD.3D 11Ø PRINT USING 1ØØ:8.5E+2ØØ
84	The IMAGE format string was deleted during the PAGE FULL interrupt routine.
85	A portion of the IMAGE format string was deleted or altered during the PAGE FULL interrupt routine.
86	A portion of the data specified in the PRINT statement was deleted during the PAGE FULL interrupt routine.
87	A data item specified in the PRINT USING statement is too large to fit into the print field specified in the format string. Example: 1ØØ IMAGE 5A 11Ø PRINT USING 1ØØ: "HORSE FEATHERS" In this example, the string constant "HORSE FEATHERS" is too large to fit into the 5 character field which is specified in line 1ØØ.

MESSAGE NUMBER	ERROR MESSAGE
88	Not used.
89	A ROM pack has issued an error message.
90	Not used.
91	Not used.
92	Not used.
93	Not used.
94	Not used.
95	An internal conversion error has occurred because a parameter in the specified statement is negative.
96	An internal conversion error has occurred because a parameter in the specified statement is greater than 65535.

SYSTEM ERROR Ø

A system error Ø is generated whenever a firmware failure condition occurs. An INIT command and a DELETE ALL command are issued to recover program control.

An example of a firmware failure condition is when a program is stored on magnetic tape in binary format from a 4052 Graphic System, and the program contains commands which are not available on the 4051 Graphic System. When such a program is loaded into the 4051 Graphic System, the firmware fails and a system error Ø results.

Appendix B

TABLES

DEFAULT I/O ADDRESSES

APPEND	@ 33,4:
BRIGHTNESS	@ 32,30:
CHARSIZE	@ 32,17:
CLOSE	@ 33,2:
COPY	@ 32,10:
DASH	@ 32,31:
DRAW	@ 32,20:
FIND	@ 33,27:
FONT	@ 32,18:
GIN	@ 32,24:
HOME	@ 32,23:
INPUT	@ 31,13:
KILL	@ 33,7:
LIST	@ 32,19:
MARK	@ 33,28:
MOVE	@ 32,21:
OLD	@ 33,4:
PAGE	@ 32,22:
PRINT	@ 32,12:
RDRAW	@ 32,20:
READ	@ 34,14:
RMOVE	@ 32,21:
SAVE	@ 33,1:
SECRET	@ 37,29:
TLIST	@ 32,19:
WRITE	@ 33,15:

ASCII Character Value Chart

Decimal Value	ASCII Character Symbol and Name	Binary Value
ø	@ (NUL Null)	0000000
1	A (SOH Start of Heading)	0000001
2	B (STX Start of Text)	0000010
3	C (ETX End of Text)	0000011
4	D (EOT End of Transmission)	0000100
5	E (ENQ Enquiry, also known as Who-Are-You)	0000101
6	F (ACK Acknowledge)	0000110
7	G (BEL Bell)	0000111
8	H (BS Backspace)	0001000
9	I (HT Horizontal Tab)	0001001
10	J (LF Line Feed)	0001010
11	K (VT Vertical Tab)	0001011
12	L (FF Form Feed)	0001100
13	(CR Carriage Return)	0001101
14	N (SO Shift Out)	0001110
15	O (SI Shift In)	0001111
16	P (DLE Data Link Escape)	0010000
17	Q (DC1 Device Control 1)	0010001
18	R (DC2 Device Control 2)	0010010
19	S (DC3 Device Control 3)	0010011
20	T (DC4 Device Control 4)	0010100
21	U (NAK Negative Acknowledge)	0010101
22	V (SYN Synchronous Idle)	0010110
23	W (ETB End of Transmission Block)	0010111
24	X (CAN Cancel)	0011000
25	Y (EM End of Medium)	0011001
26	Z (SUB Substitute)	0011010
27	[(ESC Escape)	0011011
28	' (FS File Separator)	0011100
29	] (GS Group Separator)	0011101
30	↑ (RS Record Separator)	0011110
31	_ (US Unit Separator)	0011111
32	SP (Space, Blank)	0100000
33	!	0100001
34	"	0100010
35	#	0100011
36	$	0100100
37	%	0100101
38	&	0100110
39	'	0100111
40	(	0101000
41	)	0101001
42	*	0101010

ASCII Character Value Chart (Cont)

Decimal Value	ASCII Character Symbol and Name	Binary Value
43	+	0101011
44	,	0101100
45	–	0101101
46	.	0101110
47	/	0101111
48	ø (Zero)	0110000
49	1	0110001
50	2	0110010
51	3	0110011
52	4	0110100
53	5	0110101
54	6	0110110
55	7	0110111
56	8	0111000
57	9	0111001
58	:	0111010
59	;	0111011
60	<	0111100
61	=	0111101
62	>	0111110
63	?	0111111
64	@	1000000
65	A	1000001
66	B	1000010
67	C	1000011
68	D	1000100
69	E	1000101
70	F	1000110
71	G	1000111
72	H	1001000
73	I	1001001
74	J	1001010
75	K	1001011
76	L	1001100
77	M	1001101
78	N	1001110
79	O (oh)	1001111
80	P	1010000
81	Q	1010001
82	R	1010010
83	S	1010011
84	T	1010100
85	U	1010101
86	V	1010110

ASCII Character Value Chart (Cont)

Decimal Value	ASCII Character Symbol and Name	Binary Value
87	W	1010111
88	X	1011000
89	Y	1011001
90	Z	1011010
91	[(Left Bracket)	1011011
92	\ (Reverse Slash)	1011100
93	] (Right Bracket)	1011101
94	↑ (Up Arrow)	1011110
95	_ (Underscore)	1011111
96	` (Accent Grave)	1100000
97	a	1100001
98	b	1100010
99	c	1100011
100	d	1100100
101	e	1100101
102	f	1100110
103	g	1100111
104	h	1101000
105	i	1101001
106	j	1101010
107	k	1101011
108	l	1101100
109	m	1101101
110	n	1101110
111	o	1101111
112	p	1110000
113	q	1110001
114	r	1110010
115	s	1110011
116	t	1110100
117	u	1110101
118	v	1110110
119	w	1110111
120	x	1111000
121	y	1111001
122	z	1111010
123	{ (Left Brace)	1111011
124	| (Vertical Bar)	1111100
125	} (Right Brace)	1111101
126	~ (Tilde)	1111110
127	↓ (Down Arrow)	1111111

ASCII Character Priority for String Inequalities

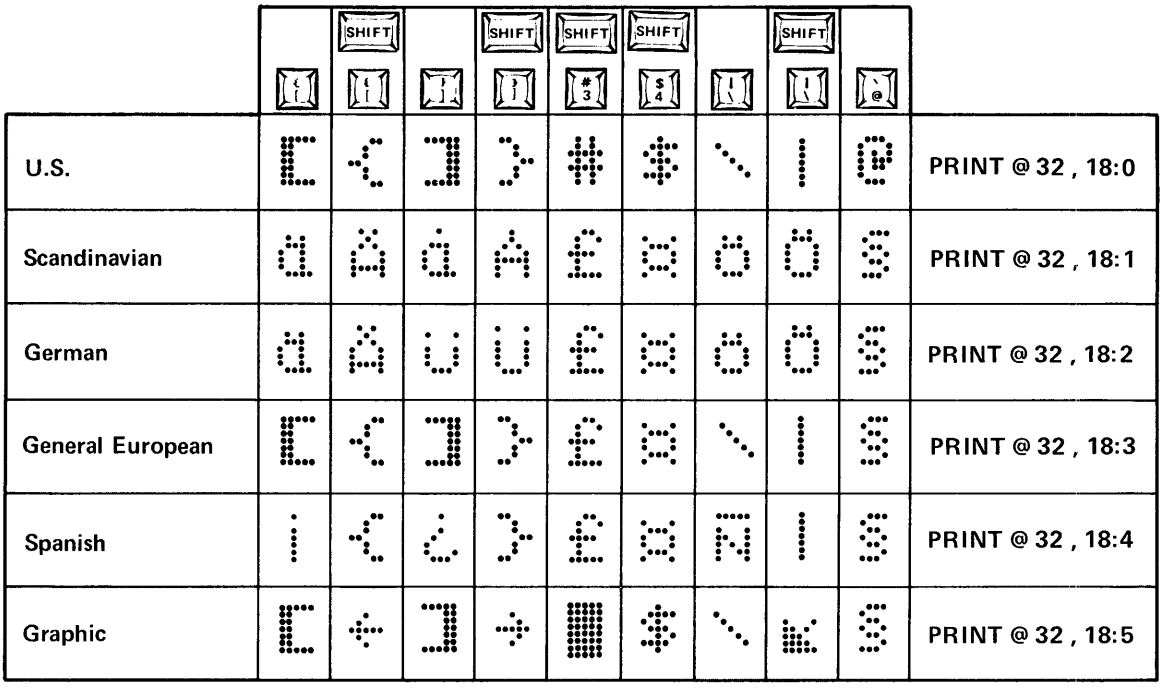
HIGHEST PRIORITY		
↓ (Down Arrow)	C or c	SP (Space, Blank)
~ (Tilde)	B or b	US (Unit Separator)
| (Vertical Bar)	A or a	RS (Record Separator)
` (Accent Grave)	@	GS (Group Separator)
_ (Underscore)	?	FS (File Separator)
↑ (Up Arrow)	=	ESC (Escape)
] (Right Bracket or Brace)	;	SUB (Substitute)
\ (Reverse Slash)	:	EM (End of Medium)
[(Left Bracket or Brace)	9	CAN (Cancel)
Z or z	8	ETB (End of Transmission Block)
Y or y	7	SYN (Synchronous idle)
X or x	6	NAK (Negative Acknowledge)
W or w	5	DC4 (Device Control 4)
V or v	4	DC3 (Device Control 3)
U or u	3	DC2 (Device Control 2)
T or t	2	DC1 (Device Control 1)
S or s	1	DLE (Data Link Escape)
R or r	Ø (Zero)	SI (Shift In)
Q or q	/	SO (Shift Out)
P or p	.	CR (Carriage Return)
O or o	—	FF (Form Feed)
N or n	,	VT (Vertical Tab)
M or m	+	LF (Line Feed)
L or l	*	HT (Horizontal Tab)
K or k	)	BS (Backspace)
J or j	(	BEL (Bell)
I or i	'	ACK (Acknowledge)
H or h	&	ENQ (Enquire, also known as Who-Are-You)
G or g	%	EOT (End of transmission)
F or f	$	ETX (End of Text)
E or e	#	STX (Start of Text)
D or d	"	SOH (Start of Heading)
	!	NUL (Null)
		LOWEST PRIORITY

NOTE: If NOCASE is set, lower case letters have a higher priority over their equivalent upper case letters (i.e., "w" > "W" is true).

Control Character Chart for GS Display

Control Character	Keyboard Input	Displayed Character	Function Performed
BEL (BELL)	CTRL G	G	Rings bell
BS (Backspace)	CTRL H	H	Backspaces the cursor
HT (Horizontal tab)	CTRL I	I	Tabs cursor to next tab stop
LF (Linefeed)	CTRL J	J	Moves cursor down one line
VT (Vertical tab)	CTRL K	K	Moves cursor up one line
FF (Form feed)	CTRL L	L	Erases screen and moves cursor up to Home
CR (Carriage Return)	CTRL M	Does not display character	Performs same function as RETURN key
RS (Record Separator)	CTRL ↑	↑	Returns the cursor to the HOME position
US (Unit Separator)	CTRL RUBOUT	—	Carriage Return, Line Feed

4051



Character Font Chart for GS Display

4052 and 4054

	SHIFT #3	Ø	@	H	N	J	SHIFT [	SHIFT \	SHIFT]	
ASCII	#	Ø	@	H	N	J	[	\	]	FONT 0
SWEDISH	#	Ø	@	H	N	J	Ä	Ö	Å	FONT 1
GERMAN	£	Ø	@	H	N	J	Ä	Ö	Ü	FONT 2
BRITISH	£	Ø	@	H	N	J	[	\	]	FONT 3
SPANISH	#	Ø	@	H	N	J	[	\	]	FONT 4
GRAPHIC	#	Ø	@	H	N	J	÷	×	÷	FONT 5
RESERVED	Same as FONT 0									FONT 6
RESERVED	Same as FONT 0									FONT 7
BUSINESS	£	Ø	@	H	N	J	[	\	]	FONT 8
DANISH	#	Ø	@	H	N	J	[	\	]	FONT 9

Numeric Error Conditions

Numeric Error Conditions				
MATH OPERATION	CAUSE OF ERROR	EXAMPLE	NUMBER RETURNED	ERROR TYPE
Addition (+) Subtraction (–) Multiplication (*) Division (/)	Parameter too Large or too Small	1E2ØØ*1E2ØØ	+ ∞	SIZE
		–1E2ØØ*1E2ØØ	– ∞	SIZE
		1/1E2ØØ	Ø	NO ERROR
	Division by Zero	4/Ø	+ ∞	SIZE
		– 4/Ø	– ∞	SIZE
Exponentiation (↑)	Parameter too Large or too Small	1E2ØØ↑1E2ØØ	+ ∞	SIZE
		–1E2ØØ↑1E2ØØ	– ∞	SIZE
	A<Ø and B not an integer in the range Ø to 255	– 2 ↑ 3	– 8	NO ERROR
		– 2 ↑ 6.5	+ ∞	FATAL
Square Root	Negative parameter	SQR(–4)	2	SIZE
Sine X	| X | ⩾ 4.116E+5 (radians)	SIN (4.2E+5)	Ø	SIZE
Cosine X	| X | ⩾ 4.116E+5 (radians)	COS (4.2E+5)	Ø	SIZE
Tangent X	| X | ⩾ 4.116E+5 (radians)	TAN (4.2E+5)	Ø	SIZE
TAN 9Ø°	Parameter Out of Range	SET DEG TAN (9Ø)	– ∞	NO ERROR
		SET DEG TAN (–9Ø)	+ ∞	NO ERROR
e^x	Parameter Out of Range	EXP (71Ø)	+ ∞	SIZE
		EXP (–71Ø)	Ø	SIZE
Matrix Inversion	Determinant is Ø	INV X	Undetermined Answer	SIZE
Matrix Multiply	Floating Point Overflow	A MPY B	Answers + ∞ – ∞	SIZE

General Purpose Interface Bus Addresses

GPIB PRIMARY ADDRESSES																	
PERIPHERAL DEVICE NUMBER	PRIMARY LISTEN ADDRESS									PRIMARY TALK ADDRESS								
	DECIMAL VALUE	DIO BUS								DECIMAL VALUE	DIO BUS							
		8	7	6	5	4	3	2	1		8	7	6	5	4	3	2	1
Device Ø	32	0	0	1	0	0	0	0	0	64	0	1	0	0	0	0	0	0
Device 1	33	0	0	1	0	0	0	0	1	65	0	1	0	0	0	0	0	1
Device 2	34	0	0	1	0	0	0	1	0	66	0	1	0	0	0	0	1	0
Device 3	35	0	0	1	0	0	0	1	1	67	0	1	0	0	0	0	1	1
Device 4	36	0	0	1	0	0	1	0	0	68	0	1	0	0	0	1	0	0
Device 5	37	0	0	1	0	0	1	0	1	69	0	1	0	0	0	1	0	1
Device 6	38	0	0	1	0	0	1	1	0	7Ø	0	1	0	0	0	1	1	0
Device 7	39	0	0	1	0	0	1	1	1	71	0	1	0	0	0	1	1	1
Device 8	4Ø	0	0	1	0	1	0	0	0	72	0	1	0	0	1	0	0	0
Device 9	41	0	0	1	0	1	0	0	1	73	0	1	0	0	1	0	0	1
Device 1Ø	42	0	0	1	0	1	0	1	0	74	0	1	0	0	1	0	1	0
Device 11	43	0	0	1	0	1	0	1	1	75	0	1	0	0	1	0	1	1
Device 12	44	0	0	1	0	1	1	0	0	76	0	1	0	0	1	1	0	0
Device 13	45	0	0	1	0	1	1	0	1	77	0	1	0	0	1	1	0	1
Device 14	46	0	0	1	0	1	1	1	0	78	0	1	0	0	1	1	1	0
Device 15	47	0	0	1	0	1	1	1	1	79	0	1	0	0	1	1	1	1
Device 16	48	0	0	1	1	0	0	0	0	8Ø	0	1	0	1	0	0	0	0
Device 17	49	0	0	1	1	0	0	0	1	81	0	1	0	1	0	0	0	1
Device 18	5Ø	0	0	1	1	0	0	1	0	82	0	1	0	1	0	0	1	0
Device 19	51	0	0	1	1	0	0	1	1	83	0	1	0	1	0	0	1	1
Device 2Ø	52	0	0	1	1	0	1	0	0	84	0	1	0	1	0	1	0	0
Device 21	53	0	0	1	1	0	1	0	1	85	0	1	0	1	0	1	0	1
Device 22	54	0	0	1	1	0	1	1	0	86	0	1	0	1	0	1	1	0
Device 23	55	0	0	1	1	0	1	1	1	87	0	1	0	1	0	1	1	1
Device 24	56	0	0	1	1	1	0	0	0	88	0	1	0	1	1	0	0	0
Device 25	57	0	0	1	1	1	0	0	1	89	0	1	0	1	1	0	0	1
Device 26	58	0	0	1	1	1	0	1	0	9Ø	0	1	0	1	1	0	1	0
Device 27	59	0	0	1	1	1	0	1	1	91	0	1	0	1	1	0	1	1
Device 28	6Ø	0	0	1	1	1	1	0	0	92	0	1	0	1	1	1	0	0
Device 29	61	0	0	1	1	1	1	0	1	93	0	1	0	1	1	1	0	1
Device 3Ø	62	0	0	1	1	1	1	1	0	94	0	1	0	1	1	1	1	0
UNLISTEN/UNTALK	63	0	0	1	1	1	1	1	1	95	0	1	0	1	1	1	1	1

General Purpose Interface Bus Addresses

GPIB Secondary Addresses									
Secondary Address	Predefined Meaning	Decimal Value	Data Bus						
			8	7	6	5	4	3	2	1
Ø	"STATUS"	96	0	1	1	0	0	0	0	0
1	SAVE	97	0	1	1	0	0	0	0	1
2	CLOSE	98	0	1	1	0	0	0	1	0
3	OPEN	99	0	1	1	0	0	0	1	1
4	OLD/APPEND	1ØØ	0	1	1	0	0	1	0	0
5	CREATE	1Ø1	0	1	1	0	0	1	0	1
6	TYPE	1Ø2	0	1	1	0	0	1	1	0
7	KILL	1Ø3	0	1	1	0	0	1	1	1
8	UNIT	1Ø4	0	1	1	0	1	0	0	0
9	DIRECTORY	1Ø5	0	1	1	0	1	0	0	1
1Ø	COPY	1Ø6	0	1	1	0	1	0	1	0
11	RELABEL	1Ø7	0	1	1	0	1	0	1	1
12	PRINT	1Ø8	0	1	1	0	1	1	0	0
13	INPUT	1Ø9	0	1	1	0	1	1	0	1
14	READ	11Ø	0	1	1	0	1	1	1	0
15	WRITE	111	0	1	1	0	1	1	1	1
16	ASSIGN	112	0	1	1	1	0	0	0	0
17	"ALPHASCALE"	113	0	1	1	1	0	0	0	1
18	FONT	114	0	1	1	1	0	0	1	0
19	LIST/TLIST	115	0	1	1	1	0	0	1	1
2Ø	DRAW/RDRAW	116	0	1	1	1	0	1	0	0
21	MOVE/RMOVE	117	0	1	1	1	0	1	0	1
22	PAGE	118	0	1	1	1	0	1	1	0
23	HOME	119	0	1	1	1	0	1	1	1
24	GIN	12Ø	0	1	1	1	1	0	0	0
25	"ALPHAROTATE"	121	0	1	1	1	1	0	0	1
26	COMMAND	122	0	1	1	1	1	0	1	0
27	FIND	123	0	1	1	1	1	0	1	1
28	MARK	124	0	1	1	1	1	1	0	0
29	SECRET	125	0	1	1	1	1	1	0	1
3Ø	"ERROR"	126	0	1	1	1	1	1	1	0
31	undefined	127	0	1	1	1	1	1	1	1

Keyword Instructions to the BASIC Interpreter

APPEND

1. Evaluate the I/O address specified in this APPEND statement. If an I/O address is not specified, use the default I/O address for APPEND. Convert the primary address to the appropriate primary talk address, then place both the primary talk address and the secondary address in temporary storage.
2. Evaluate the parameters in this statement. Prepare to assemble a program from the incoming ASCII character string.
3. Take the I/O address out of temporary storage and issue the address to the specified input device.
4. Give control to the addressed input device and prepare to receive an ASCII character string.
5. Input the ASCII character string. After each CR character check to see that the previous characters conform to the syntax of a valid BASIC statement, then place the statement into temporary program storage.
6. If the statement does not conform to the syntax of a valid BASIC statement, terminate the APPEND operation and send the invalid statement to the GS display with the appropriate error message.
7. If the input peripheral device is on the GPIB, issue an unlisten and untalk address to the peripheral device after EOI (End or Identify) is activated.
8. Insert the new program lines into the current BASIC program in the place specified by the parameters in this statement.
9. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

CLOSE

1. Evaluate the parameter in this statement (if there is one) and convert the parameter to an ASCII character. Add a Carriage Return character to the parameter and prepare to transmit. If there isn't a parameter, prepare to transmit a Carriage Return (CR).
2. Transmit the ASCII character string to the addressed device. Issue an EOI (End or Identify) signal with the CR character to tell the receiving device that the transmission is finished.
3. If the receiving device is on the GPIB, issue an untalk and unlisten command over the GPIB.
4. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

DRAW

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for DRAW. Convert the primary address to the appropriate primary listen address, then place both the primary listen address and the secondary address in temporary storage.
2. Evaluate the two parameters in this statement. If the parameters are numeric expressions, reduce the numeric expressions to numeric constants. Interpret the first parameter as the X coordinate of a new graphic data point in user data units. Interpret the second parameter as the Y coordinate of a new graphic data point in user data units. If the parameters are single dimension array variables, consider the first element in each array as the X and Y coordinates of the first graphic data point respectively; consider the second elements in each array as the X and Y coordinates of the next graphic data point, and so on.
3. Convert the X and Y coordinates from user data units to Graphic Display Units (GDU's). Refer to the parameters set up by the WINDOW and VIEWPORT statements to determine the user's data units.
4. Convert the specified X and Y coordinates to an ASCII character string using the default PRINT format as a guide. Add a Carriage Return (CR) to the end of the ASCII string and prepare to transmit.
5. Take the I/O address out of temporary storage and issue the address to the specified output device.
6. Transmit the ASCII character string to the specified output device one character at a time. If the transmission is over the GPIB, issue an EOI signal with the last CR character to tell the receiving device that the transmission is finished.
7. If the receiving device is on the GPIB, issue an untalk and unlisten command over the GPIB after the transmission if finished.
8. If operating under program control then go to the next statement; if not then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

FIND

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for FIND. Convert the primary address to the appropriate primary listen address and the secondary address in temporary storage.
2. Evaluate and reduce the parameter in this statement to a numeric constant. Make sure the numeric constant is within the range 0 to 255; if it isn't terminate the operation and print the appropriate error message on the GS display.
3. Convert the parameter to an ASCII character string. Add a Carriage Return (CR) to the ASCII character string and prepare to transmit.
4. Take the I/O address out of temporary storage and issue the address to the specified output device.
5. Transmit the ASCII character string to the specified output device one character at a time. If transmitting over the GPIB then issue an EOI (End or Identify) signal with the CR character to tell the receiving device that the transmission is finished.
6. If the receiving device is on the GPIB, issue an untalk and unlisten command over the GPIB after the transmission is finished.
7. If operating under program control, go to the next statement; if not, then monitor the keyboard for further input.

Keyword Instructions to the BASIC Interpreter

GIN

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for GIN. Convert the primary address to the appropriate primary talk address, then place both the primary talk address and the secondary address in temporary storage.
2. Prepare to receive two incoming ASCII character strings from the specified input source.
3. Take the I/O address out of storage and issue the address to the specified input device.
4. Give control to the input device.
5. Input and store ASCII characters from the input device until a Carriage Return (CR) is received or EOI is activated.
6. Interpret the first valid number in the ASCII string as the X coordinate of the current data point in GDU's (Graphic Display Units). Interpret the second valid number in the ASCII string as the Y coordinate of the current data point in GDU's. If a valid number is not found, input more ASCII characters until two valid numbers are found.
7. After two valid numbers are input, issue an unlisten and an untalk address over the GPIB if the input device is on the GPIB.
8. Convert the X and Y coordinates from GDU's to the user data values specified in the WINDOW and VIEWPORT parameters.
9. Assign the X coordinate to the first variable specified as a parameter in this statement. Assign the Y coordinate to the second variable specified as a parameter in this statement.
10. If operating under program control, go on to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

HOME

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for HOME. Convert the primary address to the appropriate primary listen address, then place both the primary listen address and the secondary address in temporary storage.
2. If the specified output device is on the General Purpose Interface Bus (GPIB), prepare to transmit a Carriage Return (CR) in ASCII code.
3. Take the I/O address out of temporary storage and issue the address to the specified output device.
4. If the output device is on the GPIB, transmit a CR character and at the same time activate the EOI (End or Identify) signal to tell the receiving device that the transmission is finished.
5. If the receiving device is on the GPIB, issue an untalk and unlisten command over the GPIB after the transmission is finished.
6. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

INPUT

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for INPUT. Convert the primary address to the appropriate primary talk address, then place both the primary talk address and the secondary address in temporary storage.
2. Evaluate the variables which are specified as parameters in this statement; determine which variables are string variables, which variables are numeric variables, which variables are array variables, and which variables are subscripted array variables.
3. Take the I/O address out of temporary storage and issue the address to the specified input device.
4. Give control to the specified input device and prepare to receive ASCII characters.
5. Input and store the ASCII characters from the input device until a Carriage Return (CR) character is received; if a percent sign (%) is specified at the beginning of the I/O address instead of an "at" sign (@), use the alternate delimiters for INPUT.
6. If the parameter in this statement is a string variable, then execute this instruction; if not, go to step 7. Assign all ASCII characters in the string to the string variable. Keep assigning characters to the string variable until a logical record separator is input or until the dimensioned size of the string variable is exceeded. If another variable is specified as a parameter in this statement, go to step 5. If not go to step 9.
7. If the parameter in this statement is a numeric variable or a subscripted array variable, execute this instruction, if not, go to step 8. Assign the first valid number in the ASCII character string to the numeric variable or to the subscripted array variable; ignore all non-numeric characters which precede the valid number; delimit on the first non-numeric character which follows the valid number. If a valid number is not found, then input more ASCII characters. If another variable is specified as a parameter in this statement, then go to step 6; if not, go to step 9.

Keyword Instructions to the BASIC Interpreter

INPUT (cont)

8. If the parameter in this statement is an array variable, then execute this instruction; if not, go to step 9. Assign the first valid number in the ASCII character string to the first element in the array; ignore all non-numeric characters which precede the valid number; delimit on the first non-numeric character which follow follows the valid number. Keep assigning numbers to the elements of the array in row major order until all the elements have an assigned value; ignore all non-numeric characters that may be imbedded between numbers. If the current ASCII character string runs out of numbers, input more ASCII characters until all the elements in the array have assigned values. If another variable is specified as a parameter in this statement, then go to step 6; if not, go to step 9.
9. If the input device is on the GPIB, issue an unlisten and an untalk address over the GPIB.
10. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

KILL

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for KILL. Convert the primary address to the appropriate primary listen address, then place both the primary listen address and the secondary address in temporary storage.
2. Evaluate and reduce the parameter in this statement to a numeric constant. Make sure the numeric constant is within the range 0 to 255; if it isn't, terminate the operation and print the appropriate error message on the GS display.
3. Add a Carriage Return character to the parameter then convert the parameter to an ASCII character string.
4. Take the I/O address out of temporary storage and issue the address to the specified output device.
5. Transmit the ASCII character string to the specified output device one character at a time. If transmitting over the GPIB then issue an EOI (End or Identify) signal with the last CR character to tell the receiving device that the transmission is finished.
6. If the receiving device is on the GPIB, issue an untalk and unlisten command over the GPIB after the transmission is finished.
7. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

LIST

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for LIST. Convert the primary address to the appropriate primary listen address then place both the primary listen address and the secondary address in temporary storage.
2. Evaluate the parameters in this statement to determine which portion of the current BASIC program to list. If parameters are not specified, prepare to list the entire program. If one parameter is specified, prepare to list the statement with that line number. If two parameters are specified, prepare to list all the statements between the line numbers; also include the statements with the specified line numbers.
3. Convert the statements to be listed into ASCII character strings. Replace all ASCII control characters with the proper LETTER – BACKSPACE – UNDERSCORE characters. Add a Carriage Return (CR) to the end of each statement.
4. Take the I/O address out of temporary storage and issue the address to the specified output device.
5. Transmit the statements to be listed as one ASCII character string to the output device. If transmitting over the GPIB, issue an EOI signal with the last CR character to tell the output device that the transmission is finished.
6. If the receiving device is on the GPIB, issue an untalk and unlisten command over the GPIB after the transmission is finished.
7. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

MARK

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for MARK. Convert the primary address to the appropriate primary listen address then place both the primary listen address and the secondary address in temporary storage.
2. Evaluate and reduce the parameters in this statement to numeric constants; if the first parameter is not within the range 1 to 255, terminate this operation and print the appropriate error message on the GS display.
3. Convert the parameters to an ASCII character string using the default PRINT format as a guide. Add a Carriage Return (CR) to the end of the ASCII string and prepare to transmit.
4. Take the I/O address out of temporary storage and issue the address to the specified output device.
5. Transmit the ASCII character string to the specified output device one character at a time. If transmitting over the GPIB, issue an EOI (End or Identify) signal with the last CR character to tell the output device that the transmission is finished.
6. If the output device is on the GPIB, issue an untalk and unlisten command over the GPIB after the transmission is finished.
7. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

MOVE

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for MOVE. Convert the primary address to the appropriate primary listen address then place both the primary listen address and the secondary address in temporary storage.
2. Evaluate the two parameters in this statement. If the parameters are numeric expressions, reduce the numeric expressions to numeric constants. Interpret the first parameter as the X coordinate of a new graphic data point in user data units. Interpret the second parameter as the Y coordinate of a new graphic data point in user data units. If the parameters are array variables, consider the first element in each array as the X and Y coordinates of the first graphic data point respectively; consider the second elements in each array as the X and Y coordinates of the next graphic data point, and so on.
3. Convert the X and Y coordinates from user data units to Graphic Display Units (GDU's). Refer to the parameters set up by the WINDOW and VIEWPORT statements to find out what the user's data units are.
4. Convert the specified X and Y coordinates to an ASCII character string using the default PRINT format as a guide. Add a Carriage Return (CR) to the end of the ASCII string and prepare to transmit.
5. Take the I/O address out of temporary storage and issue the address to the specified output device.
6. Transmit the ASCII character string to the specified output device one character at a time. If the transmission is over the GPIB, issue an EOI signal with the last CR character to tell the output device that the transmission is finished.
7. If the output device is on the GPIB, issue and untalk and unlisten command over the GPIB after the transmission is finished.
8. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

REV A, MAR 1979

Keyword Instructions to the BASIC Interpreter

OLD

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for OLD. Convert the primary address to the appropriate primary talk address, then place both the primary talk address and the secondary address in temporary storage.
2. Delete the current BASIC program and all variables from memory.
3. Take the I/O address for this statement out of temporary storage and issue the address to the specified input device.
4. Give control to the specified input device and prepare to receive ASCII character strings.
5. Input ASCII characters from the input device until a Carriage Return (CR) is transmitted. Evaluate the ASCII character string just received. Make sure the string conforms to the syntax of a valid BASIC statement; if not, terminate this operation and send the string and the appropriate error message to the GS display. If the string conforms to the BASIC language syntax, convert to program format and store in memory.
6. Repeat step 5 until the input device transmits an EOI signal with a Carriage Return (CR), an end of text character, or until memory overflows. If memory overflows, print the appropriate error message on the GS display.
7. If operating under program control, execute the new program starting with the lowest line number; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

PAGE

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for PAGE. Convert the primary address to the appropriate primary listen address then place both the primary listen address and the secondary address in temporary storage.
2. If the specified output device is on the General Purpose Interface Bus (GPIB), prepare to transmit a Carriage Return (CR) character in ASCII code.
3. Take the I/O address out of temporary storage and issue the address to the specified output device.
4. If the output device is on the GPIB, transmit a CR character and at the same time activate the EOI (End or Identify) signal to tell the output device that the transmission is finished.
5. If the output device is on the GPIB, issue an untalk and unlisten command over the GPIB after the transmission.
6. If operating under program control then go to the next statement; if not then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

PRINT

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for PRINT. Convert the primary address to the appropriate primary listen address then place both the primary listen address and the secondary address in temporary storage.
2. Evaluate the parameters in this statement. Reduce all numeric expressions to numeric constants. Reduce all string variables to string constants. If any numeric variables or string variables do not have assigned values, terminate this operation and send the appropriate error message to the GS display.
3. Convert the specified parameters into an ASCII character string using the PRINT USING format as a guide. If a PRINT USING format is not specified, use the default PRINT format as a guide.
4. Add a Carriage Return character to the end of the ASCII data string and prepare to transmit.
5. Take the I/O address out of temporary storage and issue the address to the specified output device.
6. Transmit the character string to the specified output device one character at a time. If the transmission is over the GPIB, issue an EOI signal with the last character to tell the output device that the transmission is finished.
7. If the output device is on the GPIB, issue an untalk and unlisten command over the GPIB after the transmission is finished.
8. If operating under program control then go to the next statement; if not then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

RBYTE

1. Prepare to receive data bytes over the General Purpose Interface Bus (GPIB).
2. Input a data byte over the GPIB. Assign the decimal equivalent of the data byte to the first numeric variable specified in this statement.
3. If the EOI signal line is active when the data byte is transfered, make the decimal value of the data byte negative.
4. If another numeric variable is specified in this statement, input another data byte over the GPIB. Assign the data byte to the next variable.
5. Repeat step 4 until all variables specified in this statement have assigned values.
6. If operating under program control, go to the next statement; if not, then monitor the keyboard for further input.

Keyword Instructions to the BASIC Interpreter

RDRAW

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for RDRAW. Convert the primary address to the appropriate primary listen address then place both the primary listen address and the second address in temporary storage.
2. Evaluate the two parameters in this statement. If the parameters are numeric expressions, reduce the numeric expressions to numeric constants. Interpret the first parameter as the X coordinate of a new graphic data point in user data units relative to the present position of the graphic point. Interpret the second parameter as the Y coordinate of a new graphic data point in user data units relative to the present position of the graphic point. If the parameters are array variables, consider the first element in each array as the X and Y coordinates of the first graphic data point respectively; consider the second elements in each array as the X and Y coordinates of the next graphic data point, and so on.
3. Convert the X and Y coordinates from user data units to Graphic Display Units (GDU's). Refer to the parameters set up by the WINDOW and VIEWPORT statements to find out what the user's data units are.
4. Convert the specified X and Y coordinates to an ASCII character string using the default PRINT format as a guide. Add a Carriage Return (CR) to the end of the ASCII string and prepare to transmit.
5. Take the I/O address out of temporary storage and issue the address to the specified output device.
6. Transmit the ASCII character string to the specified output device one character at a time. If the transmission is over the GPIB then issue an EOI signal with the last CR character to tell the output device that the transmission is finished.
7. If the output device is on the GPIB, issue an untalk and unlisten command over the GPIB when the transmission is finished.
8. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

READ

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for READ. Convert the primary address to the appropriate primary talk address then place both the primary talk address and the secondary address in temporary storage.
2. Evaluate the variables which are specified as parameters in this statement; determine which variables are numeric variables, which variables are string variables, which variables are array variables, and which variables are subscripted array variables.
3. Take the I/O address out of temporary storage and issue the address to the specified input source.
4. Give control to the input source and prepare to receive data items in machine dependent binary code.
5. Input a data item and compare the item with the first variable specified as a parameter. If the data item is numeric data in machine dependent binary code and the variable is a numeric variable or a subscripted array variable, assign the data item to the variable; if not, terminate this operation and send the appropriate error message to the GS display. If the data item is numeric data in machine dependent binary code and the variable is an array variable, assign the data item to the first element in the array; if not, terminate this operation and send the appropriate error message to the GS display. Repeat this step for each array element; assign data items to the array in row major order. If the data item is a character string in machine dependent binary code and the variable is a string variable, assign the character string to the string variable; if not, terminate this operation and send to appropriate error message to the GS display.
6. Repeat step 5 for each variable specified as a parameter.
7. If the input source is an external peripheral device, issues an untalk and unlisten address over the GPIB after the last variable has an assigned value.
8. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

REV A, MAR 1979

Keyword Instructions to the BASIC Interpreter

RMOVE

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for RMOVE. Convert the primary address to the appropriate primary listen address then place both the primary listen address and the secondary address in temporary storage.
2. Evaluate the two parameters in this statement. If the parameters are numeric expressions, reduce the numeric expressions to numeric constants. Interpret the first parameter as the X coordinate of a new graphic data point in user data units relative to the present position of the graphic point. Interpret the second parameter as the Y coordinate of a new graphic data point in user data units relative to the present position of the graphic point. If the parameters are array variables, consider the first element in each array as the X and Y coordinates of the first graphic data point respectively; consider the second elements in each array as the X and Y coordinates of the next graphic data point, and so on.
3. Convert the X and Y coordinates from user data units to Graphic Display Units (GDU's). Refer to the parameters set up by the WINDOW and VIEWPORT statements to find out what the user's data units are.
4. Convert the specified X and Y coordinates to an ASCII character string using the default PRINT format as a guide. Add a Carriage Return (CR) to the end of the ASCII string and prepare to transmit.
5. Take the I/O address out of temporary storage and issue the address to the specified output device.
6. Transmit the ASCII character string to the specified output device one character at a time. If the transmission is over the GPIB, issue an EOI signal with the last CR character to tell the output device that the transmission is finished.
7. If the output device is on the GPIB, issue an untalk and unlisten command over the GPIB when the transmission is finished.
8. If operating under program control then go to the next statement; if not then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

SAVE

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for SAVE. Convert the primary address to the appropriate primary listen address then place both the primary listen address and the secondary address in temporary storage.
2. Make a copy of the statements in the current BASIC program and convert the statements to ASCII character strings. Add a Carriage Return (CR) to the end of each statement.
3. Take the I/O address out of temporary storage and issue the address to the specified output device.
4. Transmit the current program as one ASCII character string to the specified output device. If transmitting over the GPIB, issue an EOI (End or Identify) signal with the last CR character to tell the output device that the transmission is finished.
5. If the output device is on the GPIB, issue an untalk and unlisten command over the GPIB when the transmission is finished.
6. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

SECRET

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for HOME. Convert the primary address to the appropriate primary listen address, then place both the primary listen address and the secondary address in temporary storage.
2. If the specified output device is on the General Purpose Interface Bus (GPIB), prepare to transmit a Carriage Return (CR) in ASCII code.
3. Take the I/O address out of temporary storage and issue the address to the specified output device.
4. If the output device is on the GPIB, transmit a CR character and at the same time activate the EOI (End or Identify) signal to tell the output device that the transmission is finished.
5. If the output device is on the GPIB, issue an untalk and unlisten command over the GPIB after the transmission is finished.
6. If operating under program control, go to the next statement; if not, then monitor the keyboard for further input.

Keyword Instructions to the BASIC Interpreter

TLIST

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for TLIST. Convert the primary address to the appropriate primary listen address then place both the primary listen address and the secondary address in temporary storage.
2. Address the GS magnetic tape unit directly; instruct the Tape Unit to rewind the tape cartridge, FIND the first file, then send the information contained in the file header.
3. Take the I/O address out of temporary storage and issue the address to the specified output device.
4. Transfer the first file header to the output device in ASCII format.
5. Instruct the GS Magnetic Tape Unit to find and send the information contained in the next file header.
6. Transfer the file header to the specified output device in ASCII format.
7. Repeat steps 5 and 6 until the Tape Unit sends an EOT (End of Tape) signal.
8. After the EOT signal is received, issue an unlisten and untalk over the GPIB.
9. If the output device is on the GPIB, issue an untalk and unlisten command over the GPIB after the GS magnetic tape unit issues an EOT signal.
10. Instruct the GS Magnetic Tape Unit to rewind the tape cartridge.
11. If operating under program control, go to the next statement; if not, then monitor the GS keyboard for further input.

Keyword Instructions to the BASIC Interpreter

WBYTE

1. Evaluate and reduce the numeric expressions which are specified as parameters in this statement to numeric constants. Round each numeric constant to an integer and check to make sure the integer falls within the range −255 to +255; if the integer does not fall within the range, terminate this operation and print the appropriate error message on the GS display.
2. Activate the ATN (Attention) signal line on the GPIB if an "at" sign (@) is specified after the keyword WBYTE. Keep the ATN line active until a colon (:) is reached in the parameter listing.
3. Treat each parameter as the decimal equivalent of a binary data byte.
4. Transfer the data byte represented by decimal value one at a time over the GPIB. If the decimal equivalent of a data byte is negative, activate the EOI (End or Identify) signal line as the data byte is transfered.
5. If operating under program control, go to the next statement; if not, then monitor the keyboard for further input.

Keyword Instructions to the BASIC Interpreter

WRITE

1. Evaluate the I/O address specified in this statement. If an I/O address is not specified, use the default I/O address for WRITE. Convert the primary address to the appropriate primary listen address then place both the primary listen address and the secondary address in temporary storage.
2. Evaluate the parameters in this statement. Reduce all numeric expressions to numeric constants. Reduce all string variables to string constants. Prepare a two byte header for each data item indicating the type of data (numeric or character string) and the length of the item in bytes.
3. Take the I/O address out of temporary storage and issue the address to the specified output device.
4. Transmit the data items to the specified output device one byte at a time in machine dependent binary code. If the output device is on the GPIB, issue an EOI (End or Identify) signal with the last byte to tell the output device that the transmission is finished.
5. If the output device is on the GPIB, issue an untalk and unlisten command over the GPIB when the transmission is finished.
6. If operating under program control then go to the next statement; if not then monitor the GS keyboard for further input.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
Ø	96	Ø	1	1	Ø	Ø	Ø	Ø	Ø	

1. The BASIC interpreter is executing a statement which represents a change in your status parameters.
2. Prepare to receive an ASCII character string containing the parameter changes.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Change your internal parameters according to the information contained in the ASCII character string.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
1	97	Ø	1	1	Ø	Ø	Ø	Ø	1	

1. The BASIC interpreter is executing a SAVE statement.
2. Prepare to receive an ASCII character string one character at a time.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Save the information in the ASCII character string.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
2	98	Ø	1	1	Ø	Ø	Ø	1	Ø	

1. The BASIC interpreter is executing a CLOSE statement.
2. Prepare to receive an ASCII character which represents a digit from Ø through 9.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Interpret the digit as the logical unit number for the file to be closed.
5. If the digit is a Ø then close all the open files.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
3	99	Ø	1	1	Ø	Ø	Ø	1	1	

1. The BASIC interpreter is executing an OPEN statement.
2. Prepare to receive an ASCII character string which represents the parameters of the statement.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. OPEN the specified file using the information contained in the ASCII string.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
4	1ØØ	Ø	1	1	Ø	Ø	1	Ø	Ø	

1. The BASIC interpreter is executing an OLD or APPEND statement.
2. Send ASCII characters starting at the present position of the read head.
3. Send an EOI (End or Identify) signal when the end of the file is reached.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
5	1Ø1	Ø	1	1	Ø	Ø	1	Ø	1	

1. The BASIC interpreter is executing a CREATE statement.
2. Prepare to receive an ASCII character string which represents the parameters of the statement.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. CREATE the specified file using the information in the ASCII character string.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
6	1Ø2	Ø	1	1	Ø	Ø	1	1	Ø	

1. The BASIC interpreter is executing a TYP function.
2. Send the proper ASCII characters to indicate the type of the next data item in the specified file.
3. If the file is empty or not open, send a Ø.
4. If the next data item is an End Of File mark, send a 1.
5. If the next data item is an ASCII character string, send a 2.
6. If the next data item is a binary numeric value, send a 3.
7. And, if the next data item is a binary character string, send a 4.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
7	1Ø3	Ø	1	1	Ø	Ø	1	1	1	

1. The BASIC interpreter is executing a KILL statement.
2. Prepare to receive an ASCII character string.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Interpret the ASCII character string as the name or number of a file to be killed.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
8	1Ø4	Ø	1	1	Ø	1	Ø	Ø	Ø	

1. The BASIC interpreter is executing an UNIT statement.
2. Prepare to receive an ASCII character string.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Interpret the ASCII character string as a disk drive unit number.
5. Switch control to the specified disk drive unit.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
9	1Ø5	Ø	1	1	Ø	1	Ø	Ø	1	

1. The BASIC interpreter is executing a DIRECTORY statement.
2. Prepare to receive the parameters of the statement as an ASCII character string.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Evaluate the ASCII string and transmit the specified directory information when addressed as a talker.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
1Ø	1Ø6	Ø	1	1	Ø	1	Ø	1	Ø	

1. The BASIC interpreter is executing a COPY statement.
2. Prepare to receive the parameters of the statement as an ASCII character string.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Execute the COPY function using the information contained in the ASCII string.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
11	1Ø7	Ø	1	1	Ø	1	Ø	1	1	

1. The BASIC interpreter is executing a RELABEL statement.
2. Prepare to receive the parameters of the statement as an ASCII character string.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Execute the RELABEL function using the information contained in the ASCII string.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
12	1Ø8	Ø	1	1	Ø	1	1	Ø	Ø	

1. The BASIC interpreter is executing a PRINT statement.
2. Prepare to receive the parameters of the statement as an ASCII character string.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
13	1Ø9	Ø	1	1	Ø	1	1	Ø	1	

1. The BASIC interpreter is executing an INPUT statement.
2. Send ASCII characters starting at the present position of the read head.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
14	11Ø	Ø	1	1	Ø	1	1	1	Ø	

1. The BASIC interpreter is executing a READ statement.
2. Send data items in the Graphic System's machine dependent binary code starting at the present position of the read head.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus								Instructions to the Peripheral Device
		8	7	6	5	4	3	2	1	
15	111	Ø	1	1	Ø	1	1	1	1	

1. The BASIC interpreter is executing a WRITE statement.
2. Prepare to receive data items in the Graphic System's machine dependent binary code.
3. Store the data items starting at the present position of the write head.

Secondary Address	Decimal Value	Data Bus								Instructions to the Peripheral Device
		8	7	6	5	4	3	2	1	
16	112	Ø	1	1	1	Ø	Ø	Ø	Ø	

1. The BASIC interpreter is executing an ASSIGN statement.
2. Prepare to receive an ASCII character string which represents the parameters of the statement.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Execute the ASSIGN function using the information contained in the ASCII string.

Secondary Address	Decimal Value	Data Bus								Instructions to the Peripheral Device
		8	7	6	5	4	3	2	1	
17	113	Ø	1	1	1	Ø	Ø	Ø	1	

1. The BASIC interpreter is preparing to send new "ALPHASCALE" parameters.
2. Prepare to receive an ASCII character string specifying the width and the height of alphanumeric characters.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Set the ALPHASCALE parameters using the information contained in the ASCII string.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
18	114	Ø	1	1	1	Ø	Ø	1	Ø	

1. The BASIC interpreter is preparing to send new FONT information.
2. Prepare to receive an ASCII string which represents the specified character font.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Switch from the current character set to the character set specified in the ASCII string.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
19	115	Ø	1	1	1	Ø	Ø	1	1	

1. The BASIC interpreter is executing a LIST or TLIST statement.
2. Prepare to receive an ASCII character string representing file header information from the Graphic System's internal Magnetic Tape Unit.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. PRINT the file header information starting at the present position of the writing tool.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
2Ø	116	Ø	1	1	1	Ø	1	Ø	Ø	

1. The BASIC interpreter is executing a DRAW or a RDRAW statement.
2. Prepare to receive an ASCII character string containing the X and Y coordinates of one or more data points in GDU's or physical device graphic units.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. DRAW vector(s) to the new X and Y coordinate(s) sequentially starting at the present position of the writing tool.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
21	117	Ø	1	1	1	Ø	1	Ø	1	

1. The BASIC interpreter is executing a MOVE or RMOVE statement.
2. Prepare to receive an ASCII character string containing the X and Y coordinates of one or more data points in GDU's or physical device graphic units.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. MOVE the writing tool to the new X and Y coordinate(s).

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
22	118	Ø	1	1	1	Ø	1	1	Ø	

1. The BASIC interpreter is executing a PAGE statement.
2. Prepare to receive an ASCII Carriage Return (CR).
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Erase the writing surface and move the writing tool to the HOME position or execute a form feed to a new page.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
23	119	Ø	1	1	1	Ø	1	1	1	

1. The BASIC interpreter is executing a HOME statement.
2. Prepare to receive an ASCII Carriage Return (CR).
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Move the writing tool to the HOME position.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
24	120	Ø	1	1	1	1	Ø	Ø	Ø	

1. The BASIC interpreter is executing a GIN (Graphic Input) statement.
2. Send the X and Y coordinates of the current graphic data point in Graphic Display Units (GDU's).
3. Send the X coordinate first as an ASCII string (most significant digit first) followed by the Y coordinate (most significant digit first).
4. Issue an EOI (End or Identify) signal with the last caracter in the string to indicate the end of the transfer.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
25	121	Ø	1	1	1	1	Ø	Ø	1	

1. The BASIC interpreter is preparing to send new "ALPHAROTATE" information.
2. Prepare to receive an ASCII character string containing the angle of rotation for the ALPHAROTATE parameter.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Set the ALPHAROTATE parameter using the information in the ASCII character string.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
26	122	Ø	1	1	1	1	Ø	1	Ø	

1. The BASIC interpreter is executing a CMD (Command) statement.
2. Prepare to receive an ASCII character string which specifies a command to be executed.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Execute the command specified in the ASCII string.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
27	123	Ø	1	1	1	1	Ø	1	1	

1. The BASIC interpreter is executing a FIND statement.
2. Prepare to receive an ASCII character string which represents a tape file number.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. Position the read/write head to the beginning of the specified magnetic tape file.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
28	124	Ø	1	1	1	1	1	Ø	Ø	

1. The BASIC interpreter is executing a MARK statement.
2. Prepare to receive an ASCII character string which represents the parameters of the statement.
3. Look for an EOI (End or Identify) signal which indicates the end of the transfer.
4. MARK the current magnetic tape into files using the information in the ASCII string.

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
29	125	Ø	1	1	1	1	1	Ø	1	

1. The BASIC interpreter is executing a SECRET statement.
2. Make the current program secret.

Predefined Secondary Addresses

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
3Ø	126	Ø	1	1	1	1	1	1	Ø	

1. The BASIC interpreter is requesting an error message or an error number.
2. Send the message in the form of an ASCII character string.
3. Activate the EOI (End or Identify) signal line as the last character in the message is sent or terminate the message with a CR character (or both).

Secondary Address	Decimal Value	Data Bus 8	7	6	5	4	3	2	1	Instructions to the Peripheral Device
31	127	Ø	1	1	1	1	1	1	1	

undefined

INTERFACING INFORMATION

Appendix C

INTERFACING INFORMATION

4050 SERIES SYSTEM BLOCK DIAGRAM DESCRIPTION

The 4050 Series Graphic System (shown on the following page) is a microcomputer system containing a processor, a Random Access Memory (RAM), and a Read Only Memory (ROM). There are three built-in peripherals which attach directly to the processor bus lines. These peripherals are the keyboard (the primary input device), the display (the primary output device), and the magnetic tape unit (a mass storage device). Two optional peripherals can also be attached via external plug connections: a Hard Copy Unit for making paper copies of display information and a Joystick which allows the keyboard operator to manually control the position of the display cursor when entering graphic information.

There are two interfaces which provide a communication channel for external peripheral devices. The General Purpose Interface provides a bit-parallel, byte-serial data path for peripheral devices such as digital X-Y Plotters, Instrumentation Systems, and Disk File Devices. This interface conforms to IEEE Standard #488-1975. The Data Communications Interface provides a bit-serial link between the Graphic System and devices such as host computers, and modems. This interface conforms to the RS-232-C standard.

Processor

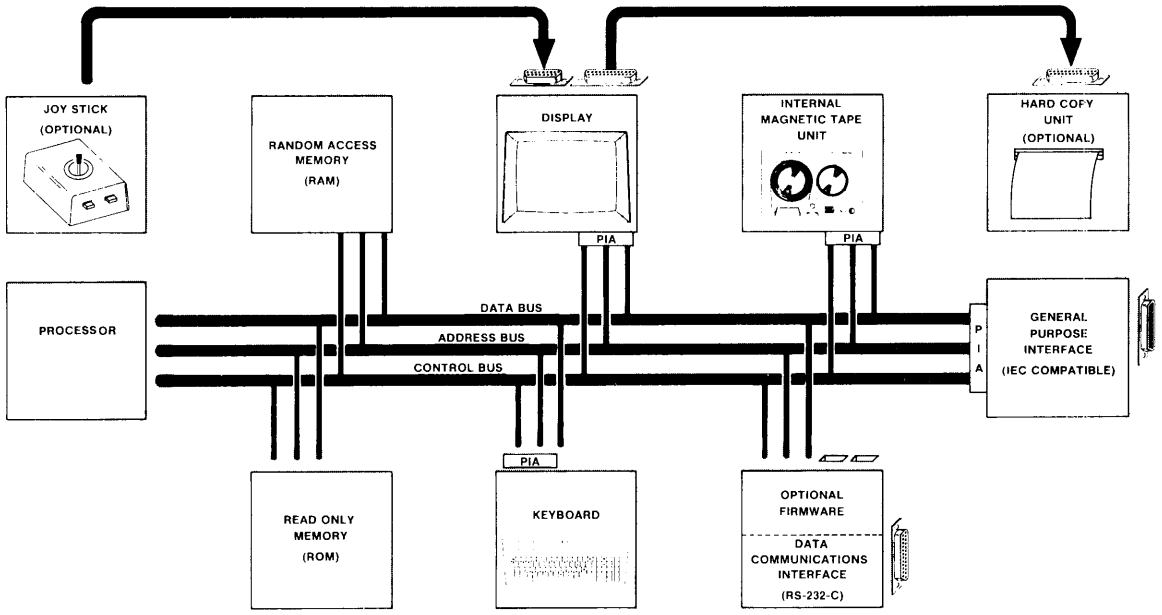
The processor is the main computing device for the system. All other modules are considered peripheral devices or support devices. The processor directs system operations as well as decodes BASIC instructions, and performs arithmetic and logic operations.

The processor is guided by a set of processor instructions which give the system the ability to "talk" in the BASIC language. These instructions are permanently fixed in the Read Only Memory (ROM).

Read Only Memory (ROM)

The Read Only Memory stores instruction codes for the processor. Each instruction is stored as an 8-bit binary code. As a whole, the processor instructions are called "firmware" because the code resembles a software program except that the code is permanently fixed in memory and cannot be changed or destroyed by turning off the system power.

The processor retrieves an instruction from the ROM by placing a 16-bit address on the Address Bus. The ROM responds by sending the instruction stored in that location to the processor over the Data Bus. The processor decodes and executes the instruction. The processor then addresses another location in ROM for the next instruction, and so on.



GRAPHIC SYSTEM BLOCK DIAGRAM

Random Access Memory (RAM)

The processor uses the Random Access Memory to temporarily store data, BASIC program instructions, and intermediate processing results for arithmetic operations. Each location stores one byte (eight bits) of information. Approximately 2000 bytes are used by the processor to store information during processing operations. This memory space is not accessible to the user. The rest of the RAM holds data and BASIC programs entered into the system from the keyboard, the magnetic tape unit, or an external peripheral device. This information is accessible to the user and can be changed, deleted, or replaced at any time. Additional RAM capacity can be added to increase storage space. The RAM, unlike the ROM, is used for temporary storage only and is erased each time power is removed from the system.

Internal Peripherals

The keyboard, display, and magnetic tape unit are internal peripherals connected to the processor bus lines with one or more Peripheral Interface Adapters (PIA's). PIA's contain registers which look like memory locations to the processor. Each PIA register has its own address within the processor's address space.

When the processor sends data to a peripheral, it addresses a PIA register in the same manner it addresses a memory location. The processor then sends data to the addressed PIA via the Data Bus as though it were sending data to a memory location. This data is held by the PIA and presented to the peripheral until the peripheral operates on the data. When the peripheral is

finished operating on the data, the peripheral signals the processor to send new data. Any previous information in the PIA register is overwritten with new data from the processor.

When the processor retrieves data from a peripheral, it retrieves the data as though it were retrieving data from a memory location. The processor places the appropriate address on the Address Bus and then activates control lines on the Control Bus which causes the PIA to place its contents on the Data Bus. The processor captures the data by placing the information in one of its own internal registers.

GS Keyboard

The Graphic System keyboard is the primary input device for the system. Each time a key is pressed a keycode is placed in the keyboard PIA. A control line on the control bus tells the processor to retrieve the keycode.

When the processor is ready, it transfers the keycode over the Data Bus to one of its own internal registers. The processor then places the keycode in a RAM location called the line buffer. A copy of the keycode is sent to the GS display where the key symbol is displayed on the screen.

The above process takes place for each keyboard entry until the operator presses the RETURN key. Before pressing the RETURN key, however, the operator has the option to make changes in the line entry by using the line editing keys. These changes are made to the contents of the line buffer. After the operator presses the RETURN key, the processor evaluates the entry and may send the results immediately to the GS display. If the entry is a BASIC statement with a line number, the processor places the entry in another part of the RAM. BASIC instructions with line numbers are not executed until the RUN statement is entered from the keyboard.

GS Display

The display is the primary output device for the system. The display is a Direct View Storage Tube (DVST) which acts like a printer. The display prints both alphanumeric and graphic information. A permanent copy of the information on the screen can be made with an attached Hard Copy Unit.

When the processor sends data to the display, it addresses a display PIA. The data is then sent to the PIA over the Data Bus. In general, this data represents a character to be printed, or a graphic coordinate to be plotted.

GS Magnetic Tape Unit

The magnetic tape unit allows the system operator to make a permanent record of information stored in the RAM. Both data and BASIC programs are transferred to and from a standard 3M ® DC—3ØØ data cartridge. Data is not directly passed from the RAM to the magnetic tape unit. Instead, the processor acts as a go-between. Information traveling to the magnetic tape unit from the RAM passes through the processor first; likewise, information traveling from the magnetic tape unit to the RAM passes through the processor first. The information transferred is merely a copy of the original contents of the RAM. The original contents of RAM are not destroyed unless power is removed from the unit or unless the contents are overwritten by new information from the keyboard or an external peripheral device.

Hard Copy Unit

The Hard Copy Unit is an optional peripheral device which makes a paper copy of information displayed on the GS display.

The Hard Copy Unit is connected directly to the GS display via a 15-pin connector located on the rear panel of the main chassis. When a MAKE COPY key is pressed (either on the GS keyboard or on the Hard Copy Unit), or when the COPY statement is executed under program control, the Hard Copy Unit takes control of the GS display circuitry. A scan is made of the entire screen for displayed information. During the scan, the processor is prevented from making any data transfers to the display. When the scan is complete, the Hard Copy Unit returns control of the display to the processor and the Hard Copy Unit ejects a paper copy of the displayed information.

Joystick

The Joystick is an external peripheral device which allows the keyboard operator to manually control the position of a graphic cursor. The graphic cursor is displayed when the POINTER statement is executed from the GS keyboard or under program control. The keyboard operator moves the cursor to any point on the display by rotating the Joystick. The position of the cursor is recorded by the BASIC program when the operator presses a keyboard key. This method of inputting the position of a graphic data point is used in interactive graphic programs.

General Purpose Interface

The General Purpose Interface allows the processor to talk or to listen to any external device which has Input/Output compatibility with the standards of the IEEE Standard #488-1975 document which describes a byte-serial, bit-parallel interface system for programmable measurement instruments. The General Purpose Interface is a standard part of every Graphic System.

The General Purpose Interface transfers data from the processor Data Bus to the General Purpose Interface Bus (GPIB). Both data busses are eight bits wide.

The processor has the freedom to send data over the GPIB in either ASCII code or machine dependent binary code. The processor establishes a data format, and assigns GPIB "listeners" and "talkers" before the actual data transfer begins.

Optional Firmware

Special processor service routines can be added to the system externally through the use of a standard rear panel firmware backpack. The backpack provides space for two plug-in ROM packs. Each ROM pack contains extra memory space for special purpose firmware. This feature allows the system to be tailored to individual user applications.

Only one ROM pack can be used by the processor at any one time. The ROM pack is electrically "bank-switched" into the processor address space by executing a CALL statement.

Data Communications Interface

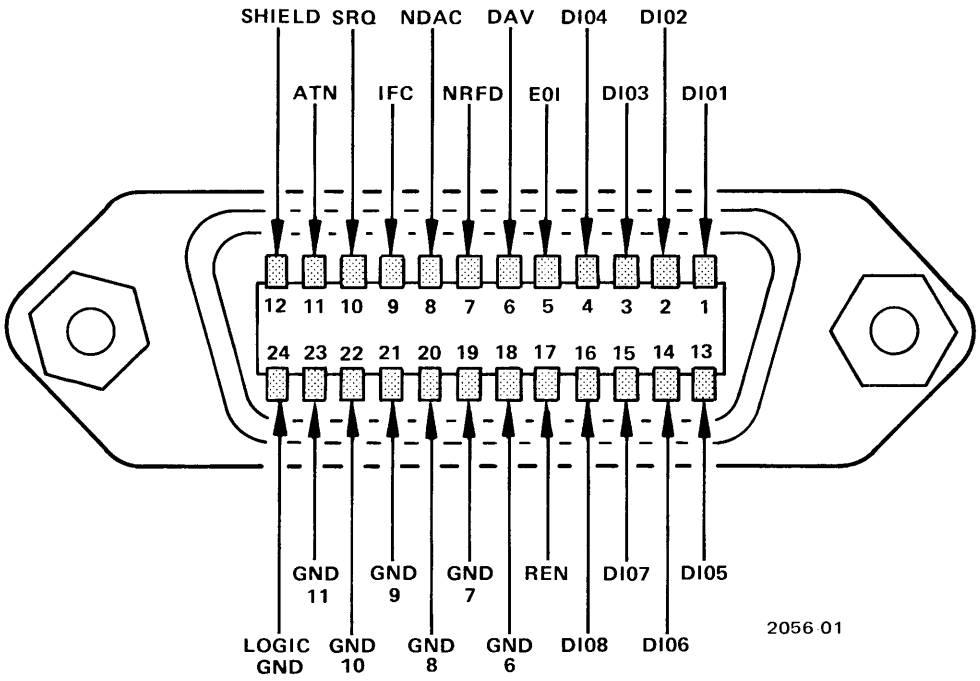
A Data Communications Interface can be purchased as an addition to the standard rear panel firmware backpack. This interface is compatible with the RS-232-C standard and uses special instructions. This allows the processor to communicate directly with devices such as display terminals, teletypes, and host computers. All of the features of the interface are completely programmable and are set by executing BASIC statements directly from the GS keyboard or under program control.

The General Purpose Interface Bus (GPIB)

The GPIB Connector

The GPIB connector is located on the rear panel of the Graphic System main chassis. This connector allows external peripheral devices to be connected to the system. The devices must conform to IEEE Standard #488-1975 which describes a byte-serial, bit-parallel interface system for programmable measuring apparatus. The GPIB connector is a standard 24-pin connector such as an Amphenol Micro-Ribbon ® connector, with sixteen active signal lines and eight interlaced grounds. The cable attached to the GPIB connector must be no longer than 2Ø meters maximum with no more than fifteen peripheral devices connected at one time.

The connector pin arrangement and signal line nomenclature is shown below:

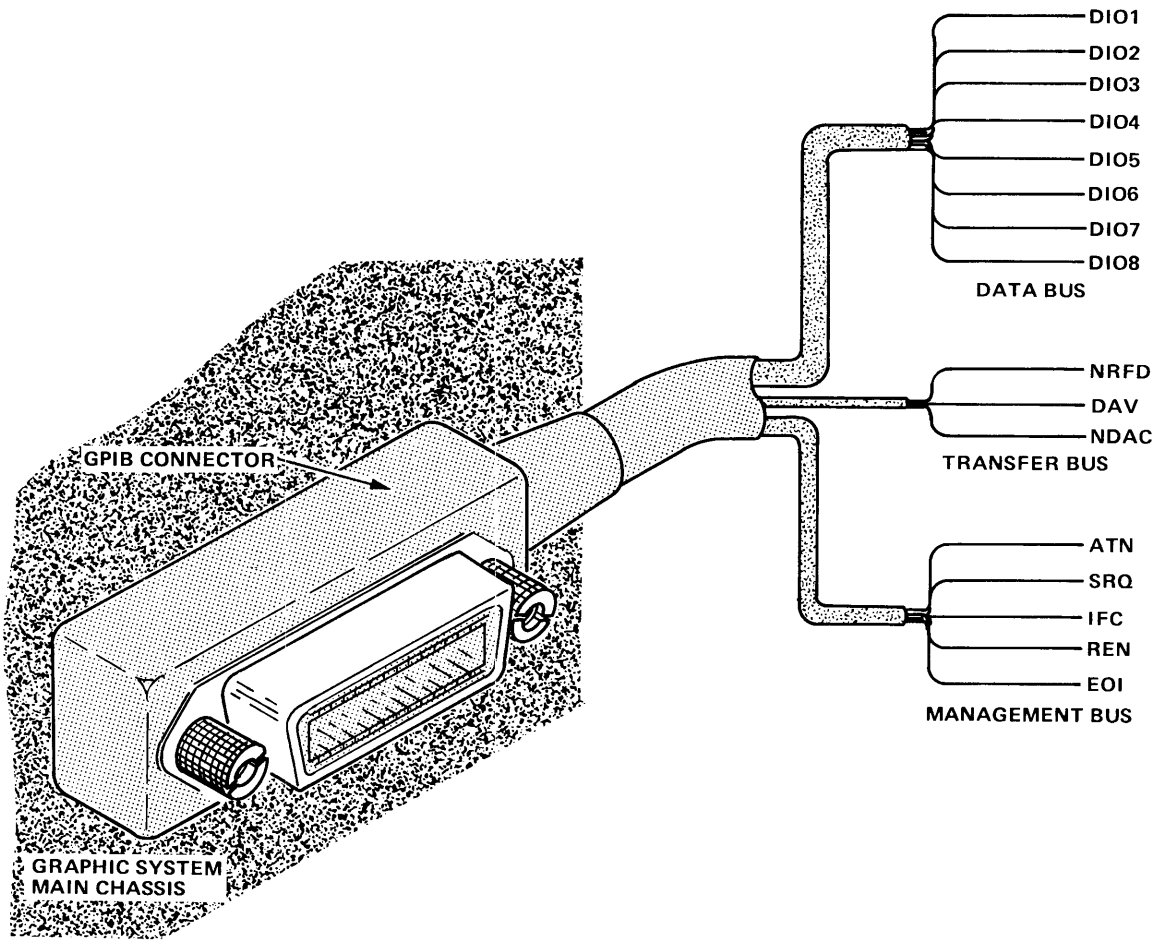


CAUTION

According to the IEEE GPIB Standard: If several devices are connected to the GPIB bus, one more than 50% of the devices must be turned on (regardless of whether they are actually used), or the GPIB may be loaded down by the turned-off devices, causing a spurious SRQ signal on the bus.

The GPIB Interfacing Concept

The GPIB is functionally divided into three component busses; an eight-line Data Bus, a three-line Transfer Bus, and a five-line Management Bus for a total of sixteen active signal lines. This bus structure is shown in the diagram below:



The transfer rate over the Data Bus is a function of the slowest peripheral device taking part in a transfer at any one time. The bus operates asynchronously with a maximum transfer rate of 250K bytes/second (one megabyte/second with tristate drivers). Both peripheral addresses and data are sent sequentially over the Data Bus. Once peripheral addresses are established for a particular transfer, successive data bytes may be transmitted in a burst for higher effective data rates.

Peripheral Devices on the GPIB are designated as talkers and listeners. The Graphic System acts as the controller to assign peripheral devices on the bus as listeners and talkers. The Graphic System further assumes that it is the only controller on the bus and it has complete control over the direction of all data transfers. There is no provision in the Graphic System for other devices on the GPIB to take turns as controller-in-charge.

A talker is a device capable of transmitting information on the Data Bus. There can be only one talker at a time. The Graphic System's processor has the ability to assume the role of the talker when it is programmed to do so.

A listener is a device capable of receiving information transmitted over the Data Bus. There may be up to fourteen listeners taking part in an I/O operation at any one time. The Graphic System's processor has the ability to assume the role of a listener any time it is programmed to do so.

GPIB Signal Definitions

Data Bus

The Data Bus contains eight bidirectional active-low signal lines, DI01 through DI08. One byte of information (eight bits) is transferred over the bus at a time. DI01 represents the least significant bit in the byte; DI08 represents the most significant bit in the byte. Each byte represents a peripheral address (either primary or secondary), a control word, or a data byte. Data bytes can be formatted in ASCII code, with or without parity (the Graphic System assumes no parity), or they can be formatted in machine dependent binary code.

Management Bus

The Management Bus is a group of five signal lines which are used to control data transfers over the Data Bus. The signal definitions for the Management Bus are as follows:

Signal	Definition
Attention (ATN)	This signal line is activated by the controller when peripheral devices are being assigned as listeners and talkers. Only peripheral addresses and control messages can be transferred over the Data Bus when ATN is active low. After ATN goes high, only those peripheral devices which are assigned as listeners and talkers can take part in the data transfer. The Graphic System assumes it is the only source of this signal.

Service Request (SRQ)	Any peripheral device on the GPIB can request the attention of the controller by setting SRQ active low. The controller responds by setting ATN active low and executing a serial poll to see which device is requesting service. This response is generated by an ON SRQ THEN statement which is executed in the BASIC program. The serial poll is taken when a POLL statement is executed in the BASIC program. After the peripheral device requesting service is found, BASIC program control is transferred to a service routine for that device. When the service routine is finished executing, program control returns to the main program. The SRQ signal line is reset to an inactive state when the device requesting service is polled. At least half of the devices connected to the GPIB must be turned on to prevent spurious SRQ signals.
Interface Clear (IFC)	The IFC signal line is activated by the controller when it wants to place all interface circuitry in a predetermined quiescent state. The Graphic System assumes that it is the only source of this signal. IFC is activated each time the INIT statement is executed in a BASIC program.
Remote Enable (REN)	The REN signal line is activated whenever the system is operating under program control. REN causes all peripheral devices on GPIB to ignore their front panel controls and operate under remote control via signals and control messages received over the GPIB.
End Or Identify (EOI)	The EOI signal can be used by the talker to indicate the end of a data transfer sequence. The talker activates EOI as the last byte of data is transmitted. When the controller is listening, it assumes that a data byte received is the last byte in the transmission, if EOI is activated. When the controller is talking, it always activates EOI as the last byte is transferred.

The Transfer Bus

A handshake sequence is executed by the talker and the listeners over the Transfer Bus each time a data byte is transferred over the Data Bus. The Transfer Bus signal lines are defined as follows:

Signal	Definition
Not Ready for Data (NRFD)	An active low NRFD signal line indicates that one or more assigned listeners are not ready to receive the next data byte. When all of the assigned listeners for a particular data transfer have released NRFD, the NRFD line goes inactive high. This tells the talker to place the next data byte on the Data Bus.
Data Valid (DAV)	The DAV signal line is activated by the talker shortly after the talker places a valid data byte on the Data Bus. An active low DAV signal tells each listener to capture the data byte presently on the Data Bus. The talker is inhibited from activating DAV when NRFD is active low.
Data Not Accepted (NDAC)	The NDAC signal line is held active low by each listener until the listener captures the data byte currently being transmitted over the Data Bus. When all listeners have captured the data byte, NDAC goes inactive high. This tells the talker to take the byte off the Data Bus.

GPIB Data Formats

Any series of bit patterns can be transmitted over the GPIB. This allows both numeric data and alphanumeric data to be transmitted in either ASCII code or machine dependent binary code.

Transferring ASCII Data

ASCII data is transferred from the read/write Random Access Memory to a peripheral device on the GPIB using the PRINT statement. ASCII transfers in the opposite direction are executed using the INPUT statement. ASCII numeric data can be transferred in either standard (free) format or scientific format, and must be transmitted most significant digit first. Valid ASCII characters are digits Ø through 9,E,e,+,−, and decimal point. ASCII character strings can be transmitted as any sequence of valid ASCII characters except Carriage Return. Carriage Return is used as the string delimiter. All ASCII data transfers, both numeric and alphanumeric, are terminated with a Carriage Return character or by activating the EOI signal line on the Management Bus, or both. (Refer to the INPUT and PRINT statements in the Input/Output Operations section for detailed information on ASCII data transfers over the GPIB).

Transferring Machine Dependent Binary Code

The term "machine dependent binary code" refers to the internal binary format used by the Graphic System to store BASIC programs and data. Data transfers between the BASIC interpreter and an external peripheral device in machine dependent binary code are normally faster because the time it takes to convert the internal binary format to ASCII format is eliminated. Transfers between the Random Access Memory and a peripheral device in machine dependent binary code implies that the peripheral device is able to understand or store the internal binary format.

Normally, transfers of this nature are carried on between the Random Access Memory and an external mass storage device which doesn't have to understand the code.

Information transfers to and from a peripheral device on the GPIB are carried on via the WRITE statement and the READ statement. Each data item transmitted is preceded by a two-byte header which identifies the data type (numeric or alphanumeric) and the length of the data item (in bytes). (Refer to the READ statement and the WRITE statement in the I/O Operations section for details on sending binary information to a peripheral device over the GPIB).

Transferring One Data Byte at a Time

Direct access to the GPIB is made available through the WBYTE (Write Byte) statement and the RBYTE (Read Byte) statement. These two statements allow you to send any eight-bit pattern over the GPIB. Also, the WBYTE statement allows you to activate the ATN signal line to tell peripheral devices that the byte you're sending is a peripheral address or a control word and it gives you complete control over the activation of the EOI signal line except when a binary Ø is transferred. (Refer to the WBYTE and RBYTE statement in the I/O Operations section for details.)

GPIB to IEEE Compatibility

Introduction

The following text describes the interfacing compatibility of the Graphic System's General Purpose Interface Bus with the IEEE Standard #488-1975 which describes a byte-serial, bit-parallel interface system for programmable measuring apparatus.

In general, the Graphic System acts as a standard talker, listener, and controller. The controller function does not have the ability to conduct a parallel poll; it does however, have the ability to conduct a serial poll. Serial polls are taken each time the POLL statement is executed.

The Graphic System does not have the ability to transfer control to another device on the GPIB with controller capability. Therefore, the Graphic System assumes that it is the only controller on the GPIB.

GPIB Interfacing Compatibility in Detail

Reference: IEEE Standard #488-1975

The Graphic System GPIB falls into the following interface function subsets as defined in the IEEE Standard #488-1975 document:

Section 2.3 SH (Source Handshake Function)
SH1—completely compatible

Section 2.4 AH (Acceptor Handshake Function)
AH1—completely compatible

Section 2.5 T (Talker Function)
TE3—basic extended talker, however, the Graphic System addresses itself internally and not over the GPIB.

Section 2.6 L (Listener Function)
LE1—basic extended listener, however, the Graphic System addresses itself internally and not over the GPIB.

Section 2.7 SR (Service Request Function)
SRØ—no capability to issue SRQ

Section 2.8 RL (Remote Local)
RLØ—no compatibility

Section 2.9 PP (Parallel Poll Function)
PPØ—no capability

Section 2.1Ø DC (Device Clear Function)
DCØ—no capability

Section 2.11 DT (Device Trigger Function)
DTØ—no capability

Section 2.12 C (Controller Function)
C1 —System Controller
C2 —Send IFC and take charge
C3 —Send REN
C4 —Respond to SRQ
C28—Send Interface Messages

Appendix D

GLOSSARY

TERM	DEFINITION
Accumulator	A temporary storage area used for storing a number, summing it with another number, and replacing the first number with the sum.
Algorithm	A step-by-step method for solving a given problem.
Argument	A value operated on by a function or a keyword. Also called a parameter.
Arithmetic Operator	Operators which describe arithmetic operations, such as +, −, *, /, ↑.
Array	A collection of data items arranged in a meaningful pattern. In the Graphic System, arrays may be one or two dimensional; that is, organized into rows, or rows and columns.
Array Variable	A name corresponding to a (usually) multi-element collection of data items. Array variables may be named with the characters A through Z and AØ through Z9.
ASCII Code	A standardized code of alphanumeric characters, symbols, and special "control" characters. ASCII is an acronym for American Standard Code for Information Interchange.
Assignment Statement	A statement which is used to assign, or give, a value to a variable.
BASIC	An acronym derived from Beginners All-purpose Symbolic Instruction Code. BASIC is a "high level" programming language because it uses English-like instructions.
BASIC Interpreter	A set of machine language instructions which give the system the ability to understand and execute BASIC statements. The BASIC interpreter resides in the Read Only Memory and is part of the operating system.
Binary String	A connected sequence of 1's and Ø's.
Bit	A Binary digit. A unit of data in the binary numbering system; a 1 or Ø.
Byte	A group of consecutive binary digits operated upon as a unit. One ASCII character, for example, is represented by one binary byte.
Character String	A connected sequence of ASCII characters, sometimes referred to as simply "string".

TERM	DEFINITION
Clipping	Removing vectors or portions of vectors which lie outside the defined window.
Coding	The process of preparing a list of successive computer instructions for solving a specific problem. Coding is usually done from a flowchart or algorithm.
Concatenate	To join together two character strings with the concatenation operator (&) forming a larger character string.
Constant	A number that appears in its actual numerical form. In the following expression, 4 is a constant: X = 4 * P
CRT	An abbreviation for Cathode Ray Tube. In the Graphic System, the CRT is a "storage" display, as opposed to a "refreshed" or TV-like display.
Cursor	The flashing rectangular image on the Graphic System display that is located at the position of the "next" character to be printed.
Debug	The process of locating and correcting errors in a program; also, the process of testing a program to ensure that it operates properly.
Default	The property of a computer that enables it to examine a statement requiring parameters, to see if those parameters are present; and, finding none, assigning substitute values for those parameters. Default actions provide a powerful means for saving memory space and time when loading program statements into memory.
Delimiter	A character that fixes the limits, or bounds, of a string of characters.
Dyadic	Refers to an operator having two operands.
Execute	To perform the operations indicated by a statement or group of statements.
Expression	Refers to either numeric expressions or string expressions. A collection of variables, constants, and functions connected by operators in such a way that the expression as a whole can be reduced to a constant.
Fatal Error	An error which causes program execution to terminate.
Flowchart	A programming tool that provides a graphic representation of a routine to solve a specific problem.

TERM	DEFINITION
Function	A special purpose operation referring to a set of calculations within an expression, as in the sine function, square root function, etc.
Grad	One grad equals 1/1ØØ of a right angle.
Graphic Display Unit (GDU)	An internal unit of measure representing one one-hundredth of the vertical axis on the graphic drawing surface.
Graphic Point	The tip of the writing tool on a graphic device (i. e. the tip of the pen on an X-Y plotter or the writing beam on the GS display).
Graphics	Computer output that is composed of lines rather than letters numbers, and symbols.
Hardware	The physical devices and components of a computer.
Index	A number used to identify the position of a specific quantity in an array or string of quantities. That is, in the array A, the elements are represented by the variables A(1), A(2), . . . A(5Ø); the indexes are 1, 2, . . . 5Ø.
Input	Data that is transferred to the Graphic System memory from an external source.
Instruction	A line number plus a statement (i.e., A line number plus a keyword plus any associated parameters).
Integer	A whole number; a number without a decimal part.
Interrupt	To cause an operation to be halted in such a fashion that it can be resumed at a later time.
Iterate	To repeatedly execute a series of instructions in a loop until a condition is satisfied.
Justify	To align a set of characters to the right or left of a reference point.
Keyboard	The device that encodes data when keys are pressed.
Keyword	An alphanumeric code that the Graphics System recognizes as a function to be performed.
Line Number	An integer establishing the sequence of execution of lines in a program. In the Graphic System, line numbers must be in the range of 1 through 65,535.
Logic	In the Graphic System, the principle of truth tables, also, the interconnection of on-off, true-false elements, etc., for computational purposes.

TERM	DEFINITION
Logical Expression	A numeric expression using the logical operators AND, OR, and NOT. The numeric expression is arranged in such a way that the numeric result is a logical 1 or a logical Ø. A logical expression may be part of a larger numeric expression involving relational operators and/or arithmetic operators.
Logical Operator	Operators which return logical 1's and Ø's, specifically, the AND, OR, and NOT operators. "True" operations return "1", "false" operations return "Ø".
Loop	Repeatedly executing a series of statements for a specified number of times. Also, a programming technique that causes a group of statements to be repeatedly executed.
Mantissa	In scientific notation, the term mantissa refers to that part of the number which precedes the exponent. For example, the mantissa in the number 1.234E+2ØØ is 1.234.
Matrix	A rectangular array of numbers subject to special mathematical operations. Also, something having a rectangular arrangement of rows and columns.
Memory	This generally refers to the Read/Write Random Access Memory that contains BASIC programs and data, as opposed to the Read Only Memory which contains the BASIC interpreter.
Monadic	Refers to an operator that has only one operand.
Numeric Constant	Any real number that is entered as numeric data; also, the contents of a numeric variable.
Numeric Expression	Any combination of numeric constants, numeric variables, array variables, subscripted array variables, numeric functions, or string relational comparisons inclosed in parentheses, joined together by one or more arithmetic, logical, or relational operators in such a way that the expression, as a whole, can be reduced to a single numeric constant when evaluated.
Numeric Function	Special purpose mathematical operations which reduce their associated parameters (or arguments) to a numeric constant.
Numeric Variable	A variable that can contain a single numeric value. Numeric variables can be named with the characters A through Z and AØ through Z9, and can be used in numeric expressions.

TERM	DEFINITION
Operand	Any one of the quantities involved in an operation. Operands may be numeric expressions or constants. In the numeric expression A = B+4*C, the numeric variables B and C, and the numeric constant 4 are operands.
Operator	A symbol indicating the operation to be performed on two operands. That is, in the expression Z + Y, the plus sign (+) is the operator.
Output	The results obtained from the Graphic System; also, information transferred to a peripheral device.
Parameter	A quantity that may be specified as different values; usually used in conjunction with BASIC statements. For example, in the statement WINDOW –5Ø, 5Ø, –1ØØ, 1ØØ, the parameters are –5Ø, 5Ø, –1ØØ, and 1ØØ.
Peripheral Device	Various devices (Hard Copy Unit, Plotter, Magnetic Tape Drive, etc.) that are used in the Graphic System to input data output data, and store data.
Program	A sequence of instructions for the automatic solution of a problem, resulting from a planned approach.
Programming	The process of preparing programs from the standpoint of first planning the process from input to output, and then entering the code into memory.
Relational Operator	An operator that causes a comparison of two operands and returns a logical result. Comparisons that are "true" return a "1", comparisons that are "false" return a "0". The relational operators in the Graphic System are =, <>, <, >, >=, and <= .
ROM	Read Only Memory. The ROM is that portion of the system memory that can not be changed. The information in the ROM can only be read. In the Graphic System, the BASIC operating system resides in a ROM.
Scalar	A single numeric value.
Scientific Notation	A format representing numbers as a fractional part, or mantissa, and a power of 10, or characteristic, as in 1.23E45.

TERM	DEFINITION
Scissoring	Removing vectors which attempt to move the graphic point off the graphic surface.
Software	Prepared programs that simplify computer operations, such as mathematics and statistics software. Software must be re-loaded into memory each time the system power is turned on.
Statement	A keyword plus any associated parameters.
String	A connected sequence of alphanumeric characters. Often called a character string.
String Constant	A string of characters of fixed length enclosed in quotation marks; also, the contents of a string variable.
String Function	Special purpose functions that manipulate character strings and produce string constants.
String Variable	A variable that contains only alphanumeric characters, or "strings". String variables can be represented by the symbols A$ through Z$. They have a default length of 72; i.e., they can contain up to 72 characters without being dimensioned in a DIM statement.
Subroutine	A part of a larger "main" routine, arranged in such a way that control is passed from the main routine to the subroutine. At the conclusion of the subroutine, control returns to the main routine. Control is usually passed to the subroutine from more than one place in the main routine.
Subscripted Array Variable	An array variable followed by one or two subscripts, as in A(9), B3(1,2), and Z(N). The subscripts refer to a specific element within the array.
Substring	A portion of a larger string; "BC", for example, is a substring within the string "ABCD".
System	A purposeful collection of interacting components (hardware and software) forming an organized whole and performing a function beyond the capability of any one component.
Target Variable	Any variable which is specified as a target to receive incoming data or the results of an operation.
Truncate	To reduce the number of least significant digits present in a number, in contrast to rounding off. For example, the number 5 is the result of truncating the decimal part of the number 5.382.

TERM	DEFINITION
User Data Units	The units of measure the programmer elects to work with for a particular graphing application. These units are established in the WINDOW statement as a numeric range for each axis. For example the vertical axis range can be set starting at Ø "dollars" and ending at 1ØØ "dollars;" the horizontal range can be set starting at the "year" 1962 and ending at the year "1975." All coordinate values for graphic statements are specified in user data units (except VIEWPORT).
Variable	A symbol, corresponding to a location in memory, whose value may change as a program executes.
Variable Name	A name selected by the programmer that represents a specific variable. Numeric variables and array variables may be named with the characters A through Z and AØ through Z9. String variables may be named with the characters A$ through Z$.
Vector	A line drawn between two points on a graphic surface.

INDEX

TEKTRONIX®
committed to technical excellence

MANUAL CHANGE INFORMATION

PRODUCT 4050 SERIES
070-2056-01

CHANGE REFERENCE C3/679
DATE 6-25-79

CHANGE:	DESCRIPTION

TEXT CHANGES

The following table, showing ASCII Decimal Equivalents for 4052/4054 character fonts, should be shown in two places in the manual:

Pages 2-10 and B-7

CODE	ASCII DECIMAL EQUIVALENT	35	48	64	91	92	93	123	124	125
0	ASCII	#	0	@	[	\	]	{	|	}
1	Swedish	#	0	@	Å	Ö	Ä	å	ö	ä
2	German	£	0	@	Ä	Ö	Ü	ä	ö	ü
3	British	£	0	@	[	\	]	{	|	}
4	Spanish	#	0	@	¡	Ñ	¿	{	|	}
5	Graphic	#	0	S	[	\	]	÷	≤	÷
6	Reserved	Same as FONT 0								
7	Reserved	Same as FONT 0								
8	Business	£	0	@	[	\	]	{	|	}
9	Danish	#	0	@	Æ	ø	Å	æ	ø	å

1940-251

Tektronix®
COMMITTED TO EXCELLENCE

MANUAL CHANGE INFORMATION

PRODUCT 4050 Series Graphic System

CHANGE REFERENCE C4/979

MANUAL PART NO. 070-2056-01

DATE 9-17-79

TEXT CHANGES

Page B-5, ASCII Character Priority for String Inequalities

Please replace Page B-5 in your manual with the attached.

ASCII Character Priority for String Inequalities

HIGHEST PRIORITY	(continued)	(continued)
↓ (Down Arrow)	C or c	SP (Space, Blank)
~ (Tilde)	B or b	US (Unit Separator)
| (Vertical Bar)	A or a	RS (Record Separator)
` (Accent Grave)	@	GS (Group Separator)
_ (Underscore)	?	FS (File Separator)
↑ (Up Arrow)	=	ESC (Escape)
\ (Reverse Slash)	;	SUB (Substitute)
Z or z	:	EM (End of Medium)
Y or y	9	CAN (Cancel)
X or x	8	ETB (End of Transmission Block)
W or w	7	SYN (Synchronous idle)
V or v	6	NAK (Negative Acknowledge)
U or u	5	DC4 (Device Control 4)
T or t	4	DC3 (Device Control 3)
S or s	3	DC2 (Device Control 2)
R or r	2	DC1 (Device Control 1)
Q or q	1	DLE (Data Link Escape)
P or p	ø (Zero)	SI (Shift In)
O or o	/	SO (Shift Out)
N or n	.	CR (Carriage Return)
M or m	—	FF (Form Feed)
L or l	,	VT (Vertical Tab)
K or k	+	LF (Line Feed)
J or j	*	HT (Horizontal Tab)
I or i	) or] or }	BS (Backspace)
H or h	(or [or {	BEL (Bell)
G or g	'	ACK (Acknowledge)
F or f	&	ENQ (Enquire, also known as Who-Are-You)
E or e	%	EOT (End of transmission)
D or d	$	ETX (End of Text)
	#	STX (Start of Text)
	"	SOH (Start of Heading)
	!	NUL (Null)
(cont in next column)	(cont in next column)	LOWEST PRIORITY

NOTE: If NOCASE is set, priority is determined by the decimal value of the ASCII characters. Refer to the preceding ASCII Character Value Chart. The character with the higher decimal value has higher priority.

Control Character Chart for GS Display

Control Character	Keyboard Input	Displayed Character	Function Performed
BEL (BELL)	CTRL G	G	Rings bell
BS (Backspace)	CTRL H	H	Backspaces the cursor
HT (Horizontal tab)	CTRL I	I	Tabs cursor to next tab stop
LF (Linefeed)	CTRL J	J	Moves cursor down one line
VT (Vertical tab)	CTRL K	K	Moves cursor up one line
FF (Form feed)	CTRL L	L	Erases screen and moves cursor up to Home
CR (Carriage Return)	CTRL M	Does not display character	Performs same function as RETURN key
RS (Record Separator)	CTRL ↑	↑	Returns the cursor to the HOME position
US (Unit Separator)	CTRL RUBOUT	—	Carriage Return, Line Feed

4051