Tektronix®
COMMITTED TO EXCELLENCE

PLOT 50
INTRODUCTION TO PROGRAMMING IN BASIC

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

Manual Part No.
070-2058-01

First Printing DEC 1975
This Printing SEP 1978

PRODUCT PLOT 50 Introduction to Programming in Basic

This manual supports the following versions of this product: Version 1 and up

MANUAL REVISION STATUS

REV.	DATE	DESCRIPTION
@	12/75	Original Issue
REV. A	9/78	Revised page format.

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 2 (cont) **Page**

Section 3 **DIRECTIVES**

Section 4 **ARRAYS**

Section 4 (cont)

Section 5

CHARACTER STRINGS

Page

Section 6

SUBROUTINES

Section 7 **EXTENDED I/O** **Page**

Section 8 **GRAPHICS** **Page**

Appendix A **REFERENCE MATERIAL**

PREFACE

Introduction to Programming in BASIC is for the non-programmer. Presented in a step-by-step informal style, the text helps you understand both general programming concepts and specific programming instructions for the Graphic System. With this manual's help, you'll find that programming is not a sinister, mysterious art; but is instead, groups of logical and easy-to-understand procedures. In no time at all, you'll be writing Graphic System programs specifically designed to meet your requirements.

You'll have to use a language the system understands when you write programs. BASIC is the language selected for the Graphic System because of its versatility, simplicity, and general acceptance. Tektronix has extended the BASIC language to make it a rich and powerful communications link between you and the Graphic System.

The Introduction to Programming in BASIC and the Graphic System Reference manuals contain some of the same subject material. This manual teaches programming in an operational format and restricts its instruction to programming as it applies to the Graphic System as a stand-alone system. This is a simplified text that provides a fundamental approach to programming; therefore, all the keywords in the Graphic System BASIC language are not presented here. The Graphic System Reference Manual, on the other hand, is written for the experienced programmer, and it presents a complete and in-depth description of programming and BASIC in an alphabetical format. The text applies to the Graphic System as a stand-alone system or when the system is enhanced by peripheral equipment.

It is expected that the non-programmer will use this text to learn programming, then depend on the Graphic System Reference Manual as a reference.

Section 1

INTRODUCTION

THE NEED FOR A LANGUAGE

Suppose you have a hundred or so numbers, and you want to average them. You know that you will have to obtain their sum and then divide that by the "number of numbers" to obtain the average. The Graphic System can be instructed to do the task. You must provide it with the numbers that are to be worked with (data), and then tell the machine in detail what to do. In order to communicate these instructions to the machine, there must exist a language that both you and it can understand.

BASIC is one such language. There are many programming languages — some of them widely known, others relatively unknown. BASIC is a language that is in wide use today, primarily because it is relatively easy to learn. Being easy to learn, however, is no implication that the language is restricted to trivial applications.

BASIC as a Language

BASIC is a "high level" language — that is, it utilizes English-like instructions rather than binary code (1's and 0's) or obscure sounding abbreviations. As a result, it is a relatively straightforward process to give instructions to the machine. If you want it to read something from a list of data, the keyword is READ; if you want to obtain the sine of some angle, the keyword is SIN, etc.

The word "BASIC" is an acronym derived from Beginners All-purpose Symbolic Instruction Code. The language was developed in the 1960's at Dartmouth College; the Graphic System recognizes an extended version of the original BASIC.

LANGUAGE CONCEPTS

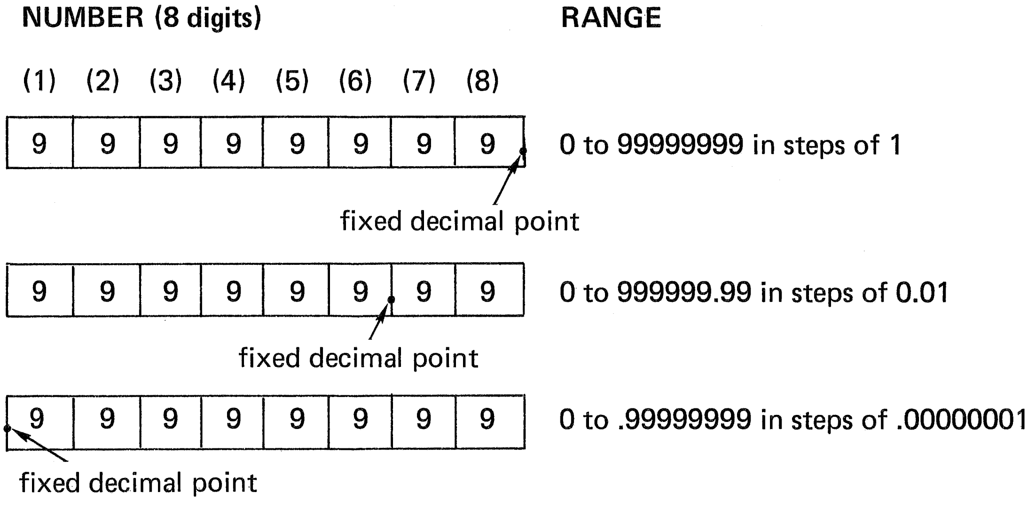
A programming language, like any language, has rules governing its use. Languages used for "people-to-people" communication are full of ambiguities and often make many assumptions. A programming language, however, requires that things be done in a precise fashion, and you cannot safely assume anything.

The English language is composed of a number of different parts. We construct a paragraph utilizing such grammatical devices as verbs, nouns, adjectives, adverbs, etc., and progress into using relatively advanced things like coordinating conjunctions, parallel construction, and so on. Programming languages also employ a kind of grammar. The particular language illustrated in this document uses (in place of verbs and nouns and similar things) "Variables", "Operators", "Statements", "Functions" and numerous other concepts which you should be familiar with before attempting to learn the language. Also, you need to be familiar with the ways that numbers can be represented. These topics are introduced in the following paragraphs.

Scientific/Standard Notation

Numbers are output from the Graphic System in two possible forms: standard notation and scientific notation. Numbers in standard notation are "human" numbers like 1.3, 950, and 0.1. Numbers in scientific notation are "machine" numbers like 1.72E+11 and 3.216E–5, where "E–5", for instance, means "times 10^{-5}". In the Graphic System, the form that a number assumes is largely determined by the magnitude of the number. Numbers within a prescribed range will be output in standard notation; numbers outside the range will be output in scientific notation.

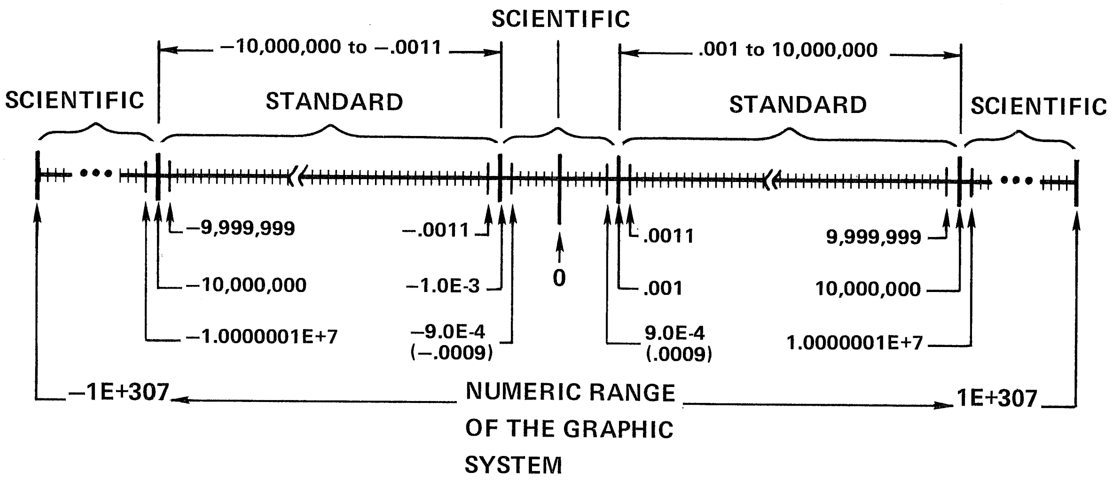
In order to visualize the range of values that numbers can assume, it may be useful to think of an arithmetic data item (number) as being a box which can contain a specific number of digits. Let's assume this box can contain eight digits. The range of values a number can take then depends on the location of the decimal point:



Notice, in the numbers used here, that the location of the decimal point not only affects the numeric range. It also affects the resolution, i.e., the smallest detectable difference between values. In standard notation, the location of the decimal point is fixed.

The smallest absolute value (excluding zero) that an eight-digit number can assume is .00000001, and the largest is 99999999. If the decimal point is fixed, however, this range cannot be attained (as the preceding illustration shows). For small numbers, the decimal must be towards the left; for large numbers, the decimal must be towards the right. The solution is to let the decimal "float." This permits the decimal point to be moved automatically by the machine, resulting in the widest possible range of values. In scientific notation, the location of the decimal point is not fixed. The points at which the Graphic System will normally perform the standard notation/scientific notation conversion is diagrammed as follows:

FIGURE 1-1



You will probably be working with standard notation most of the time.

Variables

The term "variable" is used in essentially the same context as in algebra. That is, in an equation like "X = Y + Z", X, Y, and Z are variables. One variable (in this example) is named "X", another is named "Y", etc. In BASIC, there are four kinds of variables: numeric variables, string variables, array variables, and subscripted array variables. A numeric variable is like "X" or "Y" above, and will be equal to some numeric value. The format for naming numeric varaibles is indicated as follows:

$$\left\{\begin{array}{c}\text{ONE LETTER}\\ \text{"A" through "Z"}\end{array}\right\} \quad \left[\begin{array}{c}\text{ONE NUMERAL}\\ 0\text{–}9\end{array}\right]$$

The braces indicate that you *must* use one letter of the alphabet; the square brackets indicate that the use of a numeral is optional. (This notation will be used throughout this manual. Again, braces mean that you must select one item from the list of items included within the braces; brackets mean that the enclosed items are optional.) The following are examples of valid numeric variable names:

A
X
X4
Y7

The following are examples of numeric variable names which are not valid:

AB
3P
4

An array variable looks like a numeric variable. However, array variables can be followed by one or two subscripts enclosed in parentheses.

Examples:

A(1)
Y3 (2)
Z (4,3)
X(N)

Basically, array variables allow you to refer to some specific value in an ordered group of values. For example, suppose that you have gathered five temperature readings and want to perform some calculations using these readings. The five values representing the temperatures might be stored in the machine in the form:

A(1)	A(2)	A(3)	A(4)	A(5)
65.	72.	75.	70.	67.

The array variable which refers to the entire collection is (in this case) A, and by subscripting the variable you can refer to specific elements of the collection. Thus, A(2) = 72, A(5) = 67, etc.

A *string variable* departs from the familiarities of algebraic convention. String variables are so designated because they imply equivalence to a string of characters (letters). The format for naming a string variable is as follows:

$$\left\{\begin{matrix}\text{ONE LETTER}\\ \text{"A" through "Z"}\end{matrix}\right\} \quad \left\{\ \$\ \right\}$$

Examples of invalid string variable names:

AB$
Y3$
4P$
X

With numeric variables, you can imagine a condition where, say, X = 4. With string variables, however, an example would be more like X$ = "ABCD". You will note that string variables always end with $; this is so the machine can distinguish between numeric and string variables. The uses of string variables will be elaborated upon later; generally, they exist to permit you to perform operations with text.

To summarize all this talk about variables, the important thing to remember is that there are four different types of variables that you will be concerned with. These are:

numeric variables like A and B1,
array variables like A and B,
subscripted array variables like A(1) and A(5),
string variables like A$ and X$.

Another thing to realize about variables is that each variable used within a given program requires some amount of memory, and will occupy some location in memory.

Operators

There are three categories of operators: arithmetic, relational, and logical. All involve a symbol denoting that some operation is to be performed. Four arithmetic operators are +, –, / (for divide), and * (for multiply). A fifth arithmetic operator is ↑, signifying exponentiation. The relational operators are more extensive, involving things like < (less than), > (greater than), <= (less than or equal), and so on. The logical operators are exemplified by AND, OR, and NOT. The actual use of the various operators is detailed in the next section ("ESSENTIALS OF BASIC"); the point here is that operators, like variables, are one of the components of the language. So far, the language is composed of two things: variables and operators.

Assignments

The familiar equality symbol (=) is used in programming in a manner slightly different than usual. If you say X = 4 in a programming language, it is interpreted by the machine as saying "Assign the value of 4 to the numeric variable X". The machine goes to the location in memory that houses the variable X and places a 4 in that location. Similarly, if you say Y = 2 * X this is read by the machine as "Go to the location in memory that contains X, multiply the current contents of that location by two, and assign the result to the location that contains Y". Further, X = X + 4 means "Go to the memory location containing X, add 4 to the current value of X, and place the result in the memory location X." The equals sign is sometimes termed the "assignment operator."

Functions

A function provides a means for invoking a built-in set of calculations and supplying the result. For example, if you want to set a numeric variable like X equal to the square root of 20, one way to accomplish this is by saying X = SQR (20). In this case, the square root function has been employed; the function itself is written using a name, SQR, and a number enclosed in parenthesis. The number is called the parameter of the function, and the function is said to return a value. (The parentheses are optional.) Functions, like everything else in programming, have a specific format:

NAME (PARAMETER)

The mathematical functions which are built into this version of BASIC are listed below. (For simplicity, the parameter in each case is X.)

Function	What It Returns
ABS (X)	The absolute value of X.
EXP (X)	e^X
INT (X)	Returns the "largest integer ⩽ X"; i.e., INT (–3.7) = –4 INT (3.7) = 3
LOG (X)	Logarithm of X to base e.
LGT (X)	Logarithm of X to base 10.
RND (X)	Random number between 0 and 1. If X = 0, the RND function will return the same number each time.
SQR (X)	Square root of X. (X must be positive.)
SIG (X)	Sign of X. Returns: 1 if $X > 0$ 0 if $X = 0$ –1 if $X < 0$
SIN (X)	Sine of X
COS (X)	Cosine of X
TAN (X)	Tangent of X
ATN (X)	Arc tangent of X. (ATN may also be written as ATAN)
ASN (X)	Arc sine of X
ACS (X)	Arc cosine of X
PI	The value of π. Note that no argument is required.

Some additional functions are also included in the language which pertain to string variables and matrices (arrays); they are discussed later in the appropriate sections.

Numeric Expressions

A numeric expression in BASIC is a collection of variables, functions, constants, etc., connected by arithmetic, logical, and relational operators. Such expressions are intended to cause some desired computation to occur, and can be reduced to a single numeric value. A numeric expression is exemplified as follows:

$$X = \underbrace{\text{SQR(Y) + Z * 4}}$$

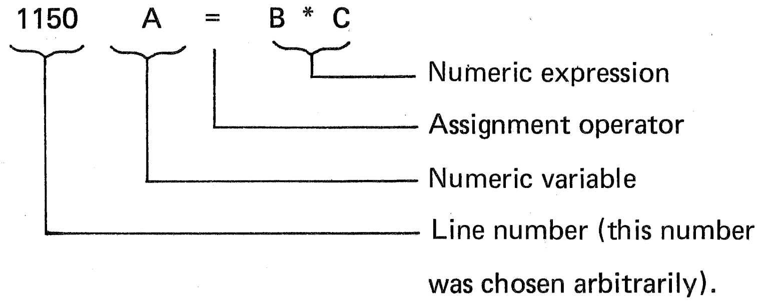
This is a numeric expression, formed with a function (SQR), the numeric variables Y and Z, the operators + and *, and the constant 4.

Statements

Up to this point, the following components of the language have been introduced:

Variables (numeric, array, subscripted array, and string)
Operators (arithmetic, relational, and logical)
Assignments
Functions
Expressions

From these components, a simple statement can be constructed:



A statement in BASIC, when used as part of a program, is always preceded with a line number. (This is sometimes termed a statement number.) Line numbers are needed so that the machine, after executing the instructions in a given line, can locate the next line in a program. The way that you signify the end of a line is by pressing the RETURN key.

A statement, then, is a valid set of instructions preceded by a line number, and ending with a "RETURN" character. A program, rather loosely defined, is a set of statements. The part of the Graphic System that processes the statements into binary code is called the BASIC interpreter. The statements in a program are not executed until the program is run. This is in contrast to what could be regarded as the "calculation mode" of the Graphic System. That is, if you just want to perform a simple calculation like obtaining the square root of 20, you do not need to create a program complete with line numbers. You just type in the expression

```
SQR (20)
```

and press RETURN. The result will appear immediately.

Section 2

ESSENTIALS OF BASIC

OPERATORS

Arithmetic Operators

Operators, as mentioned earlier, fall into three categories: arithmetic, logical, and relational. There are five arithmetic operators, tabulated as follows:

Operator	Operation	Relative Hierarchy
↑	Exponentiation	1
*	Multiplication	2
/	Division	2
+	Addition	3
−	Subtraction	3

In an arithmetic expression like A * B, the letters "A" and "B" are called *operands*. Since two operands are needed for the expression to make sense, the operator (*) is called a *dyadic* operator. A *monadic* operator, on the other hand, involves only one operand and is characterized by −B or +Y.

The column labeled "Relative Hierarchy" in the table above refers to the priority, or precedence, of the dyadic arithmetic operators. Thus, when an arithmetic expression is evaluated, exponentiation is performed first, then multiplication and division, then addition and subtraction. Equal priority operators are evaluated left-to-right. In writing arithmetic expressions, you have to remain aware of the hierarchy of the operators you are using. Also, you have to exercise care in translating algebraic expressions into BASIC. For example:

Algebraic	BASIC
4B − 3C	4 * B − 3 * C
$X^2 + 3Y^4$	X ↑ 2 + 3 * Y ↑ 4
$6X - \frac{1.2Y}{Z}$	6 * X − 1.2 * Y/Z

Note that arithmetic expressions in BASIC are represented by a single line of numbers and symbols. This can present some difficulties in translating an expression like the following:

$$\frac{4X+7}{2X-3}$$

A first attempt to convert this expression into an arithmetic expression in BASIC might yield . . .

4 * X + 7/2 * X –3

but this comes out to be interpreted as . . .

$$4X + \frac{7X}{2} - 3$$

The problem can be resolved through the use of parentheses. For example:

(4 * X + 7)/(2 * X –3)

The use of parentheses, once introduced, can be regarded as the same as in ordinary algebraic notation.

Algebraic	BASIC
$\frac{Y-3}{4Q}$	(Y – 3)/(4 * Q)
$\frac{P-14}{5R} + 3P$	(P – 14)/(5 *R) + 3 * P
$\frac{(5X-3Y)^2}{2X}$	(5 * X – 3 * Y) ↑ 2 / (2 * X)
$\sqrt{A^2+B^2}$	SQR (A ↑ 2 + B ↑ 2)

Parentheses can be used to force a different execution order in an arithmetic expression when it becomes necessary. For example, in the expression . . .

$$2X^{Y+1}$$

you might be tempted to write. . .

2 * X ↑ Y + 1

but this is equivalent to. . .

$$2X^{Y} + 1$$

The correct approach is to utilize parentheses:

2 * X ↑ (Y + 1)

A frequent error encountered in writing arithmetic expressions is that the number of left parentheses does not equal the number of right parentheses, as in . . .

((A + B)/(A –B) ↑ 2

which should be. . .

((A + B)/(A –B)) ↑ 2

Relational Operators

The BASIC language relational operators are listed below:

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

These relational operators are all dyadic operators and result in either a "true" or a "false" condition. A "true" condition is represented by a 1; a "false" condition is represented by a 0. For example, suppose that you assign numeric values to numeric variables by typing the following:

A = 5
B = 10

If you now enter the following comparison (relational) expressions, the Graphic System immediately returns the indicated results:

Comparison	Result	Meaning
A<B	1	True
A>B	0	False
A<=B	1	True
A>=B	0	False
A<>B	1	True
A=B	0	False

For example, the statement 100 N = A < B assigns the value of 1 to the variable N because 1 is the result of the comparison "A < B".

You have probably noticed that the "equals" sign at this point is capable of playing two roles: that of being an assignment operator, and that of being a relational operator.

There are two additional relational operators that differ from the relational operators mentioned above in that they do not return a "1" or "0". These are the MIN (for minimum) and MAX (for maximum) operators, as follows:

numeric expression $\begin{Bmatrix}\text{MIN}\\ \text{MAX}\end{Bmatrix}$ numeric expression

To illustrate, if X = 10 and Y = 10.7, the statement. . .

```
100 A=X MAX Y
```

assigns the value of 10.7 to the variable A. Conversely, the statement

```
110 B=X MIN Y
```

assigns 10 to the variable B. The MIN/MAX operators allow you to select the highest or lowest values from the left and right *numeric expression* once the numeric expressions are evaluated and reduced to a numeric constant. Note that these two operators, while returning the highest or lowest values, do not provide any indication of which numeric expression it was that evaluated to this value.

Logical Operators

Logical operators are similar to relational operators (excluding MIN/MAX) in that they yield 1 and 0 (true and false) results. Actually, the digits 1 and 0 are treated more as indicators of a condition or state than as a number. The 1 and 0 alternative states could be designated by any pair of opposite terms (on/off, yes/no, true/false, etc.). There are three logical operators: AND, OR, and NOT. To illustrate, let's set the numeric variables X and Y as follows:

X = 1

Y = 2

It is clear that X < Y and, conversely, Y > X. If you enter the logical expression

X < Y AND Y > X

and press RETURN, it should come as no surprise that the Graphic System returns a 1, indicating that the expression is true. The relationship to the left of the AND operator is true, the relationship to the right of the AND operator is true, and therefore the entire expression is true. With the AND operator, if both the relationship to the left and the relationship to right are true, then the expression is true. If either relationship is false, the expression is false. This characteristic of the AND operator is summarized in a "truth table" below:

Left Relationship	Right Relationship	"AND" Expression
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Again, AND expressions are true only when both components of the expression are true.

The OR operator is described by the following truth table:

Left Relationship	Right Relationship	"OR" Expression
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

As you can see, if either the left or right relationship in an OR expression is true, then the OR expression is true. This can be verified by entering the following tests (still using X = 1 and Y = 2):

```
X<Y    OR    Y>X     (true)
X<Y    OR    X>Y     (true)
X>Y    OR    X<>Y    (true)
X>Y    OR    X>=Y    (false)
```

The NOT operator differs from AND and OR in that it is a monadic operator. Its purpose is to logically negate its associated operand. The expression

NOT (1)

returns a 0; conversely, the expression

NOT (0)

returns a 1. In logical operations, any number having an absolute value of less than 0.5 is regarded as a logical 0. Any number having an absolute value of greater than 0.5 is regarded as being a logical 1.

Operator Hierarchy

The following constitutes a summary of the hierarchy of the various operators that have been presented so far:

Precedence	Operator	Elaboration
1	(	Left parenthesis
2	SIN, COS, TAN, etc.	Functions
3	Monadic + and –	Used for changing sign
4	↑	Exponentiation
5	* and /	Multiplication and division
6	Dyadic + and –	Addition and subtraction
7	MIN and MAX	Comparative operators
8	<, >, = , etc.	Relational operators
9	AND, OR, and NOT	Logical operators
10	)	Right parenthesis

A complete list of the hierarchy used in math operations is found in the Appendix.

Notice that the left parenthesis commands the highest priority of all operations; thus, the hierarchy can be altered through the use of parentheses.

ASSIGNMENT STATEMENTS

Assigning Numeric Values to Numeric Variables

An assignment statement is used to set a variable equal to some value. This is accomplished through the use of the keyword LET and the "equals" symbol (the assignment operator). The syntax form is

```
[Line number] [LET] numeric variable = numeric expression
```

where "numeric expression" can be an arithmetic expression, a constant, or another variable. This is the syntax form for a "simple" LET statement, not including, for the moment, string and array variables. Note that the keyword LET is optional, making the following two statements equivalent:

```
100 LET A=5

100 A=5
```

The reason that LET is optional is that many versions of BASIC require it in assignment statements; this way, people accustomed to using LET can continue to do so.

An assignment statement is interpreted by the Graphic System as meaning the following:

First, evaluate whatever is to the right of the equals sign.

Then, assign that value to the memory location of the variable on the left side of the equals sign.

If the variable already contains a value from a previous operation, the old value is destroyed and replaced by the new one.

Consider the following statements:

```
100 LET A=10
110 LET B=5
120 LET C=A+B
```

After the execution of these three statements, the three variables mentioned (A, B, and C) contain values as tabulated below:

VARIABLE	A	B	C
CONTENTS	10	5	15

When statement 120 (above) is executed, the current value of A (which is 10) is summed with the current value of B (which is 5), and the result becomes the current value of C. The assigned value of variables A and B remain unchanged.

A slightly different case is illustrated by the following:

```
100 LET A=5
110 LET B=10
120 LET A=A+B
```

Line 120 causes the destruction of the old value of A, replacing it with the new value which results from the summation of A and B.

INPUT

The READ and DATA Statements

Typically, you approach the Graphic System with some kind of data that requires manipulation. The objective might be to find the mean and standard deviation, or to plot the data as a graph, or to re-arrange it into a different order. In order to do any of these things, it is necessary to get the data into memory, and store it under a variable name so it can be accessed. This is made possible through two keywords, READ and DATA. READ has the following syntax form:

```
[Line number] READ numeric variable [, numeric variable] . . .
```

The "numeric variable" corresponds to the names of the variables to which the data is assigned. Later on, the READ statement will be expanded to perform extended operations; but for now this version is sufficient.

The READ statement alone is not enough to get data into memory – the keyword tells the system to "read" something, but it needs a place from which to read. For this reason, there is a DATA statement which identifies the data to be read. The syntax form is:

```
Line number DATA numeric constant [, numeric constant] . . .
```

Consider the following sequence of statements:

```
100 READ A
110 READ B
120 READ C
     .
     .
     .
200 DATA 3
210 DATA 6
220 DATA 9
```

When the BASIC interpreter encounters statement 100 during program execution, (a READ statement), it immediately searches for a DATA statement. When the first DATA statement is located, the data associated with it (3, in this case) is assigned to the variable specified in the corresponding READ statement. When the next READ statement is encountered, the BASIC interpreter looks for the next DATA statement and assigns that data to the appropriate variable. So, continuing with the above example, after line 120 is executed, the variables A, B, and C contain 3, 6, and 9, respectively.

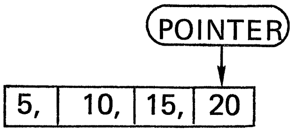
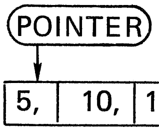
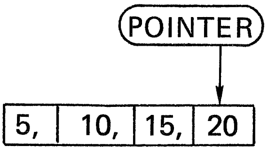
The above example can be re-written from six statements (three READs and three DATAs) to two:

```
100 READ A, B, C
      .
      .
      .
200 DATA 3, 6, 9
```

The above sequence works in the same fashion as the six line version mentioned previously. That is, the first data item is assigned to the first variable in the READ statement, and the nth data item is assigned to the nth variable specified by a READ.

Notice that the variables in the READ statement and the data items in the DATA statement are separated by commas. The commas are used as delimiters, or separators, providing the BASIC interpreter with a way to detect the end (or limit) of a particular variable name or piece of data. A DATA statement by itself, with no corresponding READ, has no effect. The DATA statement may be placed anywhere in the program. It need not immediately precede the associated READ statement.

A DATA statement, during program execution, has associated with it a "pointer" that is used to indicate which data element is to be read in next. When the first READ statement is encountered, this pointer indicates the first data item to be read in the DATA statement. A subsequent READ causes the pointer to advance to the next data item. It is possible, through the use of the keyword RESTORE, to reset the pointer to the first data item in any specified DATA statement in the program. To illustrate, consider the following tabulation of statements and their effects:

	Pointer Position after Statement Execution	**Contents of Variables after Statement Execution**
1000 DATA 5, 10, 15,20	N/A	N/A
1010 READ A, B, C, D		A: 5 E: * B: 10 F: * C: 15 G: * D: 20 H: *
1020 RESTORE		Same as above
1030 READ E, F, G, H		A: 5 E: 5 B:10 F:10 C: 15 G: 15 D: 20 H: 20

*Indicates that the variable is in an undefined state

The syntax form for RESTORE is:

[Line number] RESTORE [line number]

To reiterate, the purpose of RESTORE is to reset the data pointer to the first data item in the specified DATA statement in the program thereby "reactivating" it.

So far, you are able to input data to the Graphic System by using the READ and DATA statements, and have the option of using RESTORE. Other methods of inputting data exist, and one of these is accomplished through use of the keyword INPUT.

The INPUT Statement

The INPUT statement differs from the READ and DATA statements in that the INPUT statement allows you to enter data at program run-time. That is, when the BASIC interpreter encounters an INPUT statement during program execution, it places a blinking question mark on the screen and stops, awaiting data from the Graphic System keyboard.

The syntax form for INPUT is as follows:

```
[Line number] INPUT numeric variable [, numeric variable] . . .
```

As was the case with READ, the form of INPUT will be expanded later on in the manual.

The following two statements illustrate the use of INPUT:

```
1000 INPUT A
1010 INPUT X,Y,Z
```

When the BASIC interpreter detects an INPUT statement, the following interaction occurs:

- The BASIC interpreter generates a blinking question mark on the screen, then stops, awaiting data from the Graphic System keyboard.
- You, the operator, then enter the appropriate data from the keyboard, then signal the fact that you have done so by pressing the RETURN key.
- The BASIC interpreter then takes the data you have entered and assigns it to the variable specified in the INPUT statement.

Note from the example (statement 1010 above) that you can input data for more than one variable with a single INPUT statement by separating the variables with commas. A question mark appears for each variable specified.

OUTPUT

The PRINT Statement

With the information presented so far, you are able to input data into the system and assign it to variables. By combining the variables with operators and functions (forming expressions) you can do things to the data – addition, subtraction, etc. – but as yet there is no way to output the results. One way to obtain output is to print it on the Graphic System display; this brings up the keyword PRINT. A PRINT statement does just what you expect – it causes the assigned value of a specified variable to be printed. The syntax form is:

```
[Line number] PRINT numeric expression
```

where the numeric expression is reduced to a single numeric value. Now you can write a series of statements that begin to do something:

```
500 READ X,Y
510 DATA 100,4
520 LET Z=X/Y
530 PRINT Z
540 PRINT Z↑2
```

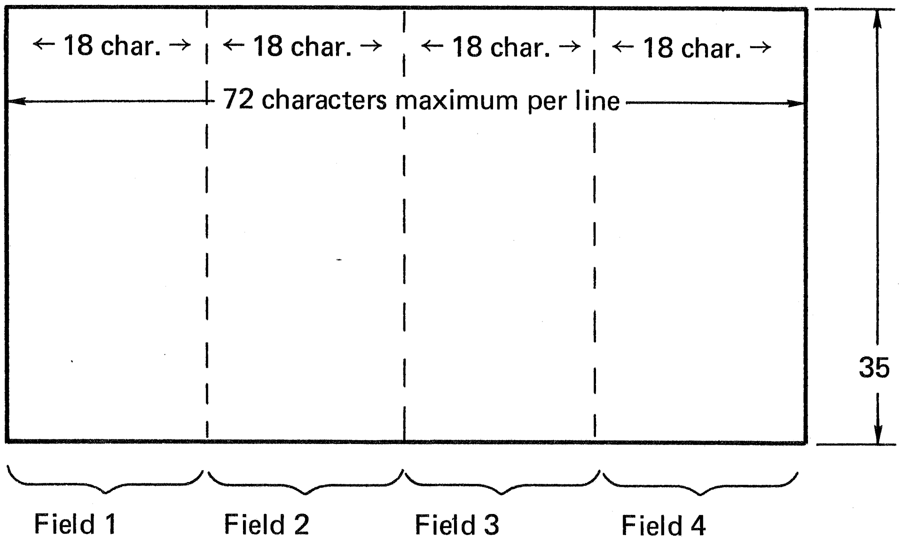
The five statements above input data, perform an operation on it, and output the result. If the five statements are executed as a complete program, the variables (X, Y, and Z) are assigned the following values:

VARIABLE	X	Y	Z
CONTENTS	100	4	25

The PRINT in statement 530 causes the current value of Z to appear on the screen, and the PRINT in statement 540 evaluates and displays Z^2.

Print Fields

The Direct View Storage Tube of the Graphic System displays 35 lines of text, each capable of containing 72 characters. An internal tab function effectively divides the display into four fields. Each field can contain up to 18 characters as shown in the following diagram.



When the BASIC interpreter executes a statement such as. . .

```
PRINT A, B, C, D
```

the comma delimiters result in the automatic generation of a tab. This tab causes the results of the next numeric expression to appear left-justified in the next print field on the display. Thus, the maximum number of values that can be printed on one line, using the comma as the delimiter, is four.

This feature provides a convenient, automatic method of formatting output into four orderly columns. For example, if a PRINT statement is followed by five variables, the fifth variable is positioned in the first print field of the next line. One restriction: a PRINT statement can not end with a comma; e.g.,

```
PRINT A, B, C, D,
```

is not allowed.

Suppressing the Automatic Tab

Occasionally it is desirable to suppress the tab function in a PRINT statement. This can be done by replacing the comma delimiter with a semicolon, as in . . .

PRINT A; B; C

Numeric variables are represented internally with a leading blank space; i.e., the number 12345, for example, can be thought of as having an internal representation of . . .

b̸ 1 2 3 4 5

where "b̸" represents a leading blank. The presence of the semicolon as a delimiter tells the BASIC interpreter to output the variable exactly as it is represented internally, starting at the present location of the cursor. The four print fields associated with the comma have no effect. The result is that a blank space separates numeric variables that are displayed in this fashion. There is an exception, however, when string constants are involved. This is discussed next.

Outputting String Constants

A string constant is any collection of alphanumeric characters enclosed in double quotes, as in

"THIS IS A STRING CONSTANT"

To output a string constant use the PRINT statement as follows:

[Line number] PRINT string constant

This opens up numerous possibilities for making your output more meaningful.

For example:

```
100 READ X,Y
110 LET S=X/Y
120 PRINT "THE ANSWER IS ";S
130 DATA 137,5
140 END
```

Note the use of the semicolon in line 120 to suppress the tab. This results in having the value of S appear right next to the printed message for best readability. Notice also the presence of a space at the end of the message in line 120. This is included for readability in the resulting output; the numeric variable "S" is not output with a leading blank, because the leading blank associated with numeric variables is suppressed when the numeric variable is preceded by a string constant in a PRINT statement. The output:

```
RUN
THE ANSWER IS 27.4
```

If quote marks appear within the string constant, there is an additional requirement. Suppose you want to print on the screen

```
THE "REAL" ANSWER
```

If you think about it, it is clear that you can not do it this way:

```
100 PRINT "THE "REAL" ANSWER"
```

The reason is that the BASIC interpreter prints out everything between the quotation marks, and they appear to end (in this case) right after the word THE. A syntax error results. The solution:

```
100 PRINT "THE ""REAL"" ANSWER"
```

To print out quotation marks, just double them. One thing to be aware of: you should never have an odd number of quotation marks in a PRINT statement.

CONTROL

Terminating Program Execution

One of the essential tasks in programming is to control the execution order of the statements in your program. Some measure of control is made possible through the fact that all statements in a program must be numbered, and they are executed in numeric sequence. An additional control feature that is required is the ability to terminate program execution. This suggests that there is a keyword in the BASIC language corresponding to this need, and indeed there is: END. The syntax form is:

```
[Line number] END
```

It is good programming practice to always use an END statement as the highest numbered statement in a program.

While END accomplishes program termination, another keyword, STOP, allows temporary breaks in program execution. One reason you might use STOP is to momentarily halt execution of a program that is not working the way you thought it would, so you can examine the current contents of variables. Or, you might want the BASIC interpreter to pause at some point in program execution. The syntax form for STOP is:

```
[Line number] STOP
```

When encountering a STOP statement, the BASIC interpreter generates the message:

```
PROGRAM STOPPED PRIOR TO LINE n
```

where n represents the statement number of the next statement beyond the STOP. To resume running the program, type RUN followed by the statement number n, then press RETURN.

The main difference between END and STOP is the fact that STOP can be thought of as a pause, while END completely terminates execution.

A Sample Program

At this point, sufficient programming techniques have been discussed to permit the writing of a complete, though not elegant, program. This particular program finds the roots of a quadratic equation of the form

$$ax^2 + bx + c = 0$$

You supply the program with values for a, b, and c, and the program calculates the values for x. These values of x become the roots of the equation, and are obtained from the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

One of the initial problems in writing a program like this is to convert the mathematical formula into instructions recognizable by the Graphic System BASIC interpreter. For instance, the "±" symbol is clearly a problem: no equivalent operator exists in the language. Also, two possible roots exist for a given quadratic equation, and the BASIC interpreter can't assign two roots to one variable, as suggested by the single "x" in the formula above.

One approach to the apparent dilemma is to make two formulae:

$$X1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

and

$$X2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

where X1 and X2 are the variables to which the roots of the equation are assigned. Now, the only remaining hurdle as far as the formulae are concerned is to rewrite them in BASIC:

```
X1  = (-B + SQR(B ↑ 2 -4 * A * C))/(2 *A)
```

and

```
X2 = (-B -SQR(B↑ 2 -4 * A * C))/(2 * A)
```

The "SQR", you will recall, is the square root function. "B ↑ 2" is "B squared". Notice that "4ac" becomes "4 * A * C", and that "2a" becomes "2 * A". The nested parentheses are evaluated algebraically with inner-most expressions being evaluated first. Notice that the parentheses around "2 * A" prevent the ambiguity which would otherwise exist: the BASIC interpreter would divide the numerator by 2, and then multiply the result by A. Remember that you must give specific instructions to the BASIC interpreter in the language it understands.

Here is the program:

```
100 PRINT "WHAT COEFFICIENT FOR A ";
110 INPUT A
120 PRINT "WHAT COEFFICIENT FOR B ";
130 INPUT B
140 PRINT "WHAT VALUE FOR C ";
150 INPUT C
160 LET X1=(-B+SQR(B↑2-4*A*C))/(2*A)
170 LET X2=(-B-SQR(B↑2-4*A*C))/(2*A)
180 PRINT
190 PRINT
200 PRINT "ROOT 1= ";X1
210 PRINT "ROOT 2= ";X2
220 END
```

The program inputs A, B, and C (statements 100—150), obtains the roots (statements 160 and 170), and outputs the results (statements 200 and 210). Once the program is entered into memory, type RUN, press RETURN, and the program responds by asking "WHAT COEFFICIENT FOR A?", and then stops. You then enter an appropriate number (like 1), and press RETURN. Values for B and C are input in the same fashion. For the sample output below, 1, 6, and 9 are used for A, B, and C.

```
RUN
WHAT COEFFICIENT FOR A 1
WHAT COEFFICIENT FOR B 6
WHAT VALUE FOR C 9


ROOT 1= -3
ROOT 2= -3
```

The 1, 6, and 9 produce the quadratic equation

$$1X^2 + 6X + 9 = 0$$

and both roots turn out to be —3. To solve a different quadratic equation, run the program again and enter the appropriate values. One thing to watch for: some quadratic equations have coefficients such that the program attempts to find the square root of a negative number (statement 160 or 170) when the quantity (B^2) is less than the quantity (4AC). This results in an error condition. There are ways around this limitation, however, and they will become apparent later in this manual.

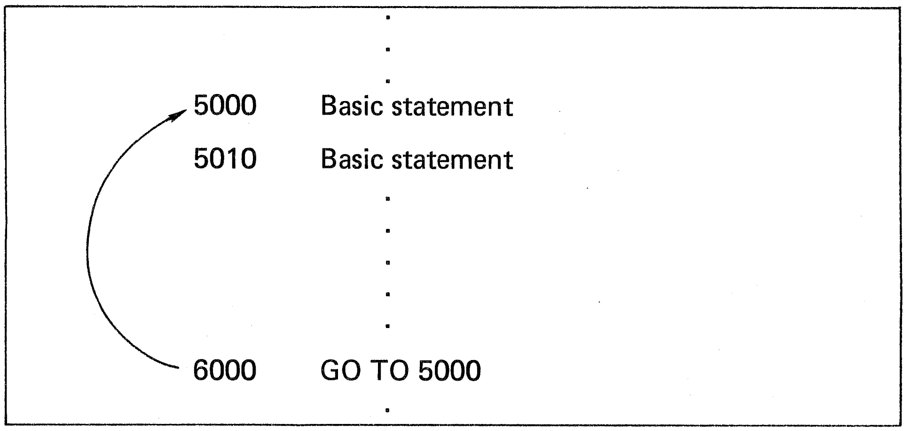
Notice how the two PRINT statements at lines 180 and 190 cause two blank lines to appear on the output. This was done to illustrate one method of adding visual organization to the output of a program. PRINT followed by nothing produces a blank line. Also, the PRINT statements at lines 100, 120, and 140 terminate with a semicolon. This suppresses the normal "carriage return" so that the question mark generated by the INPUT statements that follow appear after the printed messages.

Unconditional Transfers

One of the greatest assets inherent in programmable computing systems is the ability to repeat a set of instructions over and over at a high rate of speed. This implies that you must be able to direct the flow of program execution back to the beginning of the set of instructions which are to be repeated. In other words, you need the ability to have the BASIC interpreter go to a given statement number, execute the statement and the ones that follow, and then again go to the given statement number, etc. This leads up to a new keyword — GOTO.

```
[Line number] GOTO line number
```

GOTO, followed by a line number, directs the BASIC interpreter to "go to" the specified line number and continue executing statements from that point. This condition is known as a "loop". This looping ability allows a given set of instructions to be repeated.

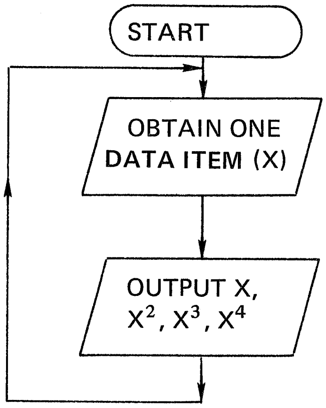


```
                 .
                 .
                 .
5000    Basic statement
5010    Basic statement
                 .
                 .
                 .
                 .
                 .
6000    GO TO 5000
                 .
```

The instruction to "GOTO" a certain statement number constitutes an unconditional transfer — program control is unconditionally transferred to the specified statement number. The brief program that follows uses a loop to input data (which in this case consists of the numbers 1 through 10), and to output each data item plus the data item squared, cubed, and quadrupled. Notice that the evaluation of the numeric expressions X^2, X^3, and X^4 is performed within the PRINT statement. Also observe that the use of commas as delimiters in the PRINT statement (line 110) produces a tabbed output, arranged into four columns.

```
100 READ X
110 PRINT X,X↑2,X↑3,X↑4
120 GO TO 100
130 DATA 1,2,3,4,5,6,7,8,9,10
140 END
```

```
RUN
 1               1               1               1
 2               4               8               16
 3               9               27              81
 4               16              64              256
 5               25              125             625
 6               36              216             1296
 7               49              343             2401
 8               64              512             4096
 9               81              729             6561
 10              100             1000            10000

DATA STATEMENT INVALID IN LINE 100 - MESSAGE NUMBER 34
```



Here is a flowchart representation of the above program. Notice the manner in which the loop is symbolized by a line. The oval-shaped symbol generally represents a start/stop point; the parallelograms represent input/output functions. The arrows represent program flow.

Incidentally, the above program is included as a demonstration only. If you run it, you will notice that an error message "DATA STATEMENT INVALID . . ." results. This is because, after inputting the final data item and generating the output, line 120 transfers program control back to line 100 where an additional READ is attempted. Since all the data has been read in, none remains, the instruction cannot be followed, and program execution terminates with the error message. The program depends on the "out of data" condition to exit the loop, which is a poor programming practice. The section which follows provides some better ways to exit loops.

Conditional Transfers and Branching

The earlier discussion about unconditional transfers employed a loop which relied on an "out of data" condition for program termination. That method was shown to be poor, but it still worked. Suppose, however, that you wanted to do a further operation after reaching an end of data condition. This suggests that there must be two paths that program execution can take: one path being the "not out of data" path, and the other being an "out of data" path. With two paths available, provision must be made to decide which path to take. The decision needs to be made based on the conditions that exist; for example, if the data is exhausted take one path; otherwise, continue with the sequence that reads in data.

This leads up to the subject of conditional transfers, and the associated BASIC statement IF . . . THEN . . .

```
[Line number] IF numeric expression THEN line number
```

Examples:

```
100 IF A<B THEN 500
200 IF X>0 THEN 650
250 IF X=-99 THEN 150
```

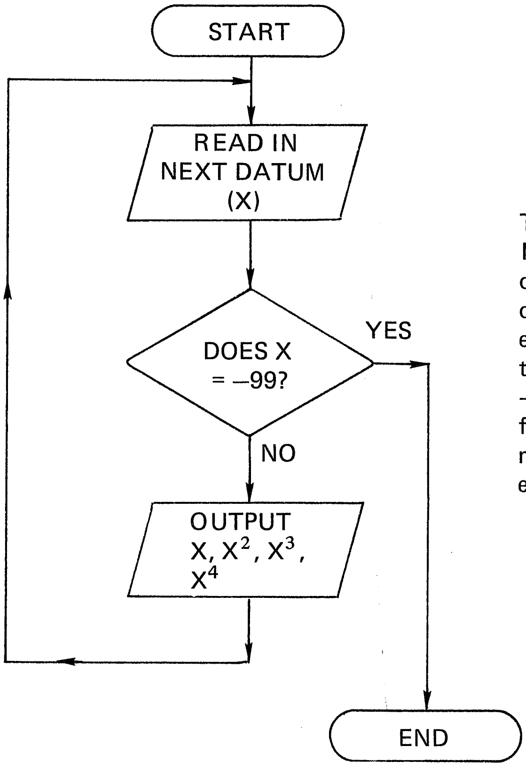
As you can see, the IF. . . THEN. . . instruction implies a "GO TO". That is, it can be viewed as saying (for example):

510 IF A = 5 THEN (GOTO) 600.

As you can see, the IF . . . THEN . . . instruction permits you to employ the decision-making capability of the BASIC interpreter. This, in turn, permits program control to be transferred to one of two paths based on the logical conditions. Consider the following short program, which is similar to the one given earlier with the discussion on unconditional transfers.

```
100 READ X
110 IF X=-99 THEN 150
120 PRINT X,X↑2,X↑3,X↑4
130 GO TO 100
140 DATA 1,2,3,4,5,6,7,8,9,10,-99
150 END
```

```
RUN
1                   1                   1                   1
2                   4                   8                   16
3                   9                   27                  81
4                   16                  64                  256
5                   25                  125                 625
6                   36                  216                 1296
7                   49                  343                 2401
8                   64                  512                 4096
9                   81                  729                 6561
10                  100                 1000                10000
```



This flowchart reveals the operation of the IF . . . THEN . . . statement in line 110. The diamond shaped symbol represents a decision, and you can see the two alternative execution paths labeled "yes" and "no". If the condition is true (if the next data item is –99), execution stops. If the condition is false, then control "falls through" the statements within the loop continue to be executed.

Notice, in statement 140, the presence of the quantity –99 included at the end of the data. This data element is a "dummy" – that is, the –99 is not intended to be incorporated as valid data, but is included as one method of marking the end of the data list. Now, examine the IF . . . THEN . . . statement at line 110. It says, "Check the current contents of X. Test to see if it equals –99. If the condition is true (if it does equal –99), then go to statement 150. If the condition is false, go to the next statement (120) and continue." The inclusion of the dummy data item, and the subsequent testing for it, provides an effective means of exiting the loop.

The term *numeric expression* in the syntax form for the IF . . . THEN . . . statement requires some elaboration. This has the form

```
numeric expression relational-operator numeric expression
```

Note that the "numeric-expression" signifies that "complex" comparisons are permitted, as in

```
A * B ↑ 2 + 3 <= D ↑ (N + 1)
```

The "relational operator" can be any one of the six such operators discussed earlier:

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

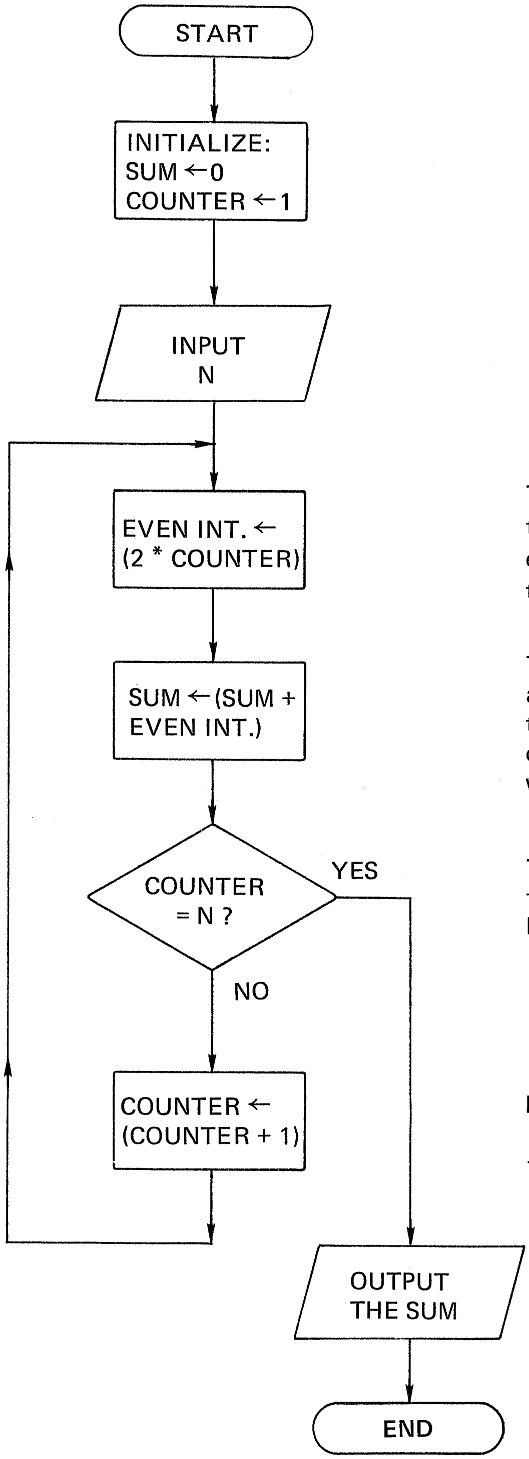
The important thing to remember about conditional transfers is that control is transferred only if the *numeric expression* is true (i.e., evaluated to a 1). Restated, a branch occurs if the condition is true. If the condition specified in the numeric expression is not true, then program control simply "falls through" to the next statement in the program.

The previous example program utilized a dummy data item and an IF . . . THEN . . . statement to exit a loop. That approach provided indirect control over the number of times the loop was executed; termination stopped when the dummy was encountered. The IF . . . THEN . . . statement can also be used to determine when to exit a loop based on the number of times the loop is repeated. To do this, you need to set up a counter to keep track of the number of times the loop repeats.

Suppose you want to devise a program that inputs an integer n, and then outputs the sum of the first n even integers. For instance, if you set n equal to five, the program outputs the sum of the first five even integers:

```
          2
          4
          6
          8
       + 10
       ----
sum =    30
```

One approach to writing such a program is to organize the routine into flowchart form, and then write the statements to perform the necessary steps. As already suggested, a requirement exists for a counter; also, the program needs a place to sum the even integers. In addition, a provision must be made to determine when n even integers have been summed. Here is a flowchart that incorporates these ingredients:



The rectangle signifies a processing function, like changing values in a variable, etc. "Sum" becomes S in the program; "counter" becomes C. These variables must be initialized so they can be used later in the routine. The arrow indicates "is assigned the value of" or "receives".

Here is where the program inputs the integer N.

The n even integers results by multiplying the current value of the counter by 2. The even integers are the numeric variable E in the program.

The sum is a running total of the even numbers added to the current value of "sum". The need for the initialization of "sum" to zero is now apparent; otherwise, the first time through the loop, "sum" would be undefined.

Test to see if the loop has repeated N times by comparing the current value of the counter with N. Branch out of the loop when the counter reaches N.

Increment the counter by adding 1 to its current value.

Print out the total. Control is not passed to this point until the loop is repeated N times.

Now that the routine is organized into flowchart form, the program can be readily produced. The output follows, using n = 5:

```
100 LET S=0
110 LET C=1
120 PRINT "HOW MANY EVEN INTEGERS ";
130 INPUT N
140 LET E=2*C
150 LET S=S+E
160 IF C=N THEN 190
170 LET C=C+1
180 GO TO 140
190 PRINT "SUM OF FIRST ";N;" EVEN INTEGERS= ";S
200 END

RUN
HOW MANY EVEN INTEGERS 5
SUM OF FIRST 5 EVEN INTEGERS= 30
```

The above exercise introduces the concept of initialization and also demonstrates how an IF . . . THEN . . . statement (or conditional transfer) is used to control the number of iterations (or successive repetitions) that are performed by a loop. In addition, the exercise illustrates some additional use of alphanumeric strings in input/output statements.

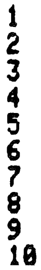
FOR/NEXT Loops

Loops have been discussed earlier, but somewhat casually. Some special provisions have been built into the language to facilitate the use of loops and to make them more convenient to employ. Remember that the use of an IF . . . THEN . . . statement to determine the number of loops entails the consecutive use of a counter, and you branch out of a loop when the counter reaches a certain limit. The whole purpose of computing systems is to make things easier, and having to manipulate counters and conditional transfers falls somewhat short of this objective.

BASIC includes a looping capability known as "FOR/NEXT loops". An example:

```
100 FOR I=1 TO 10
110 PRINT I
120 NEXT I
```

The above FOR/NEXT loop generates an output as follows:



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

FOR/NEXT loops still utilize counters, and still have a test to see if the counter has reached its limit. However, these operations are performed automatically by the BASIC interpreter. All you need to do is to specify how many times you want the loop to repeat. To repeat 5,000 times, you specify . . .

```
200     FOR I = 1 to 5000



   (statements to be repeated inside the loop)

300 NEXT I
```

If you want the loop to repeat n times, where n is defined earlier in the program, you can write:

```
  500 FOR J = 1 to N
              .
              .
              .
 (statements to be repeated)
              .
              .
              .
 610 NEXT J
```

The syntax form for FOR/NEXT loops is as follows:

```
Line number FOR numeric variable = numeric expression TO numeric expression
     [STEP numeric expression]
                    .
                    .
                    .
     (statements to be repeated)
                    .
                    .
                    .
Line number NEXT numeric variable
```

The *numeric variable* with the FOR and NEXT statements must be the same; that is, you cannot say

```
100 FOR I = 1 to 10
         .
         .
         .
150 NEXT X
```

The first two numeric expressions indicate the range through which the numeric variable is incremented. The quantity by which the numeric variable is incremented is specified by the optional protion of the FOR statement (STEP numeric expression). If you omit this part of the FOR statement, the increment, or STEP value, is assumed to be one. Restated, STEP has a default* value of one. This means that the following pair of FOR/NEXT loop program segments perform the same number of repetitions.

```
100 FOR I = 1960 to 1980
         .
         .
         .
150 NEXT I
200 FOR I = 1960 to 1980 STEP 1
         .
         .
         .
250 NEXT I
```

*In this manual, the term "default" refers to initial conditions or parameters set by the Graphic System.

If you want the numeric variable to be incremented, or "STEPed", by some quantity other than one, you specify the desired increment. This increment can be positive or negative.

Here is how a FOR/NEXT loop works:

1. When the BASIC interpreter first encounters a FOR statement, the initial *numeric expression* is evaluated, and the result is assigned to the specified *numeric variable*.

2. Next, the body of the loop (or statements to be repeated) is executed.

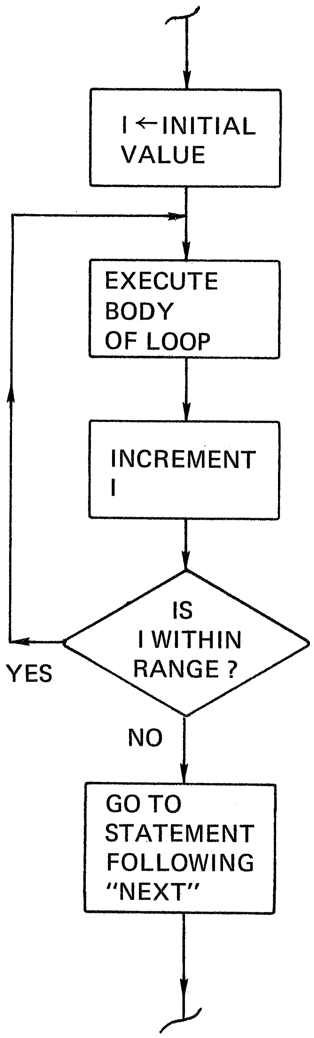
3. When the NEXT statement is encountered, the BASIC interpreter increments the *numeric variable* by an amount equal to the STEP.

4. Then, the BASIC interpreter tests to see if the current value of the *numeric variable* exceeds the range as indicated by the final *numeric expression*.

5a. If the *numeric variable* is still within range, program control is transferred back to the first statement in the body of the loop.

5b. If the *numeric variable* has been incremented to a value beyond its range, then control is transferred to the statement in the program which follows the NEXT statement.

The following flowchart summarizes the operation of the FOR/NEXT loop. The numeric variable is assumed to be "I" in this example.



FOR statement causes I to receive the value of the initial expression.

The body of the loop is always executed once.

The NEXT statement increments I by the amount indicated by the STEP.

The NEXT statement tests to see if the numeric variable is still within range. If it is, the body of the loop is exexuted again, repeatedly, until the numeric variable goes out of range. When it does, program control is transferred to the statement following the NEXT statement.

The following program tabulates the relationship between fractional inches, the decimal equivalent for each fraction, and the metric equivalent in centimeters. Two FOR/NEXT loops are employed; the first one utilizes the default step value of 1, the second uses a step value of −1.

```
100 PRINT "INCHES","DECIMAL"
110 PRINT "(FRACTIONAL)","EQUIVALENT","CENTIMETERS"
120 FOR I=1 TO 50
130 PRINT "*";
140 NEXT I
150 PRINT
160 FOR I=16 TO 1 STEP -1
170 D=I/16
180 C=D*2.54
190 PRINT I;"/16",D,C
200 NEXT I
210 END
```

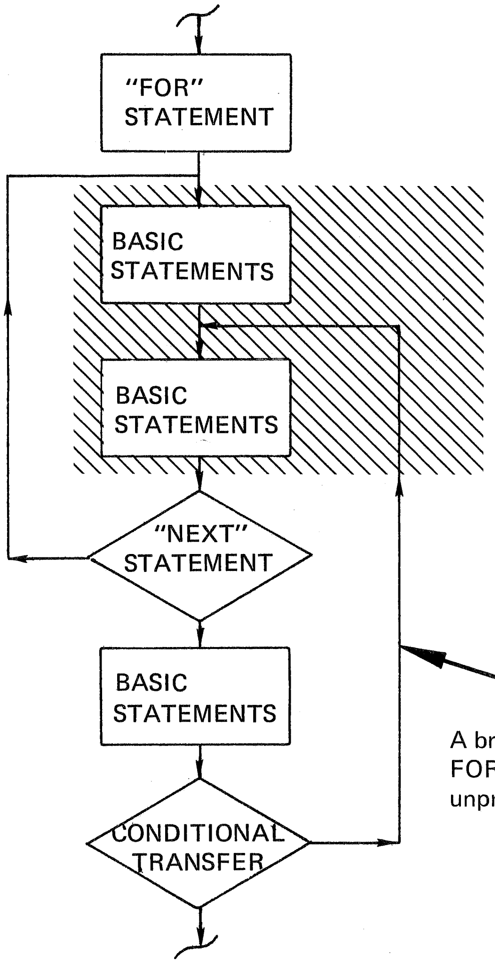
The output produced by this program is as follows:

```
INCHES            DECIMAL
(FRACTIONAL)      EQUIVALENT        CENTIMETERS
**************************************************
 16/16            1                 2.54
 15/16            0.9375            2.38125
 14/16            0.875             2.2225
 13/16            0.8125            2.06375
 12/16            0.75              1.905
 11/16            0.6875            1.74625
 10/16            0.625             1.5875
 9/16             0.5625            1.42875
 8/16             0.5               1.27
 7/16             0.4375            1.11125
 6/16             0.375             0.9525
 5/16             0.3125            0.79375
 4/16             0.25              0.635
 3/16             0.1875            0.47625
 2/16             0.125             0.3175
 1/16             0.0625            0.15875
```

The first loop (lines 120 through 140) is used only to output 50 asterisks horizontally. Statement 130 constitutes the body of the loop. The variable "I", is not really used to do anything within the body of this particular loop; it functions only as a counter. The terminating semicolon suppresses the normal "carriage return" function. The "PRINT nothing" statement 150 prints a blank character following the last asterisk, then returns the alphanumeric cursor to the beginning of the next line on the screen.

The second loop (statements 160 through 200) employs an increment of −1 for the STEP value. The first time the body of the loop is executed, I is set to 16. The second loop occurs with I equal to 15. When I finally reaches its final value (1), the loop terminates. Notice that, in this loop, the numeric variable is utilized within the body of the loop. The first column of numbers in the output is formed from a composite of the current contents of the numeric variable, and the alphanumeric string "/16". You can see the value of I decrement from 16 to 1 in this column.

One practice to avoid when using FOR/NEXT loops is branching into the body of a loop from some statement outside the loop. That is, recalling the flow diagram representation of a FOR/NEXT loop presented earlier, do not construct a program such as the following:



Beginning of FOR/NEXT loop.

Body of loop.

End of FOR/NEXT loop.

A branch such as this to a statement within a FOR/NEXT loop should be avoided because unpredictable results can occur.

Nesting FOR/NEXT Loops

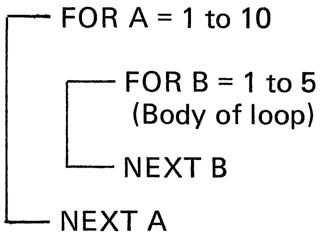
One more thing about FOR/NEXT loops: they can be *nested*. This means that the body of a FOR/NEXT loop can contain an additional FOR/NEXT loop. Consider the following program segment:

```
100 FOR I=1 TO 3
110 FOR J=1 TO 5
120 PRINT I,J
130 NEXT J
140 PRINT
150 NEXT I
```

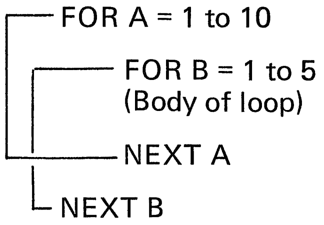
The first time this program segment is executed, I (in the "outer" loop) retains the value of 1 while J (in the "inner" loop) assumes the values of 1, 2, 3, 4, and 5. Then, I receives the value of 2, and J again becomes 1, 2, 3, 4, and 5. Finally, I becomes 3, J goes through its 5 values, and the outer loop terminates. This operation can be seen in the output produced by the above program segment:

```
1              1
1              2
1              3
1              4
1              5

2              1
2              2
2              3
2              4
2              5

3              1
3              2
3              3
3              4
3              5
```

Loops can be nested in this fashion indefinitely, and the reason for doing so will become apparent later. One thing to watch out for when nesting loops is to make sure that the loops do not "cross". That is, the control paths for nested loops should look like this:



The control paths should not look like the following:



The Computed GOTO Statement

Up to now, three main approaches to exercising control over the flow of program execution have been discussed: the simple GOTO (or unconditional transfer); the IF . . . THEN . . . (or conditional transfer); and the FOR/NEXT loop, which provides a convenient means of controlling the number of repetitions that a given program segment performs. Each of these three approaches has one thing in common: each passes program control to a single specified statement number.

Some programming tasks require that control be passed to one group of statements during one set of circumstances, to another group during a different set of circumstances, and to still a different group in other cases. For example, in processing inventory, data items bearing different types of product codes might require fundamentally different segments of the program to perform the job. One approach toward passing control to the appropriate segment of the program cound be to construct a series of IF . . . THEN . . . tests which would satisfy the requirement at the expense of using a lot of programming. This, however, is not in keeping with the attitude that computing systems exist to make things easier. Clearly, what is needed is some sort of a conditional transfer with multiple branches – and this need is fulfilled by a "computed GOTO". Specifically, in this version of BASIC, the statement that does this is a GOTO . . . OF . . . statement, with the following syntax form:

```
[Line number] GOTO numeric expression OF line number [, line number] . . .
```

When a GOTO . . . OF . . . statement is executed, the BASIC interpreter rounds off the numeric expression to an integer n, and then transfers program control to the nth statement in the line number list. If n is negative, zero, or greater than the number of statements present in the line number list, the GOTO . . . OF . . . statement is ignored. Control is passed to the statement following the GOTO . . . OF . . . statement. Also, commas must separate the statement numbers in the line number list.

Examples:

BASIC Statements	Results
500 GOTO 2 OF 750, 800, 850	Control is transferred to statement 800.
300 READ A, B, C 310 DATA 75.62, 125, 3 500 GOTO C OF 800, 900, 1000, 110	Control is transferred to statement 1000.
100 READ X, Y 110 DATA 1.3, 1.5 200 GOTO X + Y OF 400, 450, 500, 550	X + Y is rounded to 3, and control is transferred to statement 500.
100 GOTO 5 OF 200, 300, 400	Ignored. There are only three statements in the statement list.
350 X = 5 360 Y = 12 370 GOTO X – Y of 100, 200, 300	Ignored. The numeric expression evaluates to a negative integer.

DOCUMENTATION

Documenting a Program

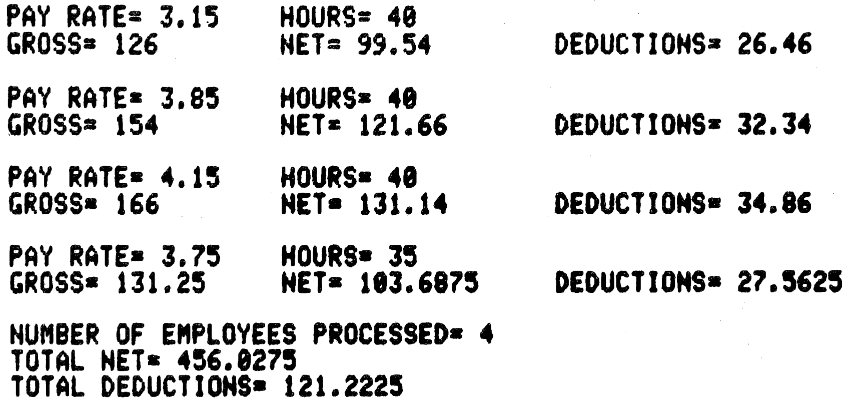
The statements in a program listing are not exactly material for light reading. In fact, lines of program statements can become so obscure that it becomes quite difficult to determine what is taking place. The problem becomes compounded when the person reading a program is not the person who generated it. Fortunately, BASIC includes the facility for incorporating information about a program ("documentation") within the program itself. Access to this facility is gained through the keyword REMARK, formed as follows:

```
[Line number] REMARK [any comments of your choosing]
```

```
100 REMARK******PAYROLL PROGRAM******
110 REMARK
120 REMARK--INITIALIZE VARIABLES
130 N0=0
140 T1=0
150 T2=0
160 REMARK--READ EMPLOYEE DATA
170 READ P,H
180 REMARK--TEST FOR DUMMY DATUM
190 IF P<0 THEN 340
200 REMARK--COMPUTE PAY
210 G=P*H
220 W=0.21*G
230 N=G-W
240 REMARK--PRINT PAYROLL
250 PRINT "PAY RATE= ";P,"HOURS= ";H
260 PRINT "GROSS= ";G,"NET= ";N,"DEDUCTIONS= ";W
270 PRINT
280 REMARK--ACCUMULATE TOTALS
290 N0=N0+1
300 T1=T1+N
310 T2=T2+W
320 REMARK--LOOP BACK TO READ MORE DATA
330 GO TO 160
340 REMARK--THIS POINT REACHED WHEN DUMMY DATUM FOUND
350 REMARK--PRINT TOTALS
360 PRINT "NUMBER OF EMPLOYEES PROCESSED= ";N0
370 PRINT "TOTAL NET= ";T1
380 PRINT "TOTAL DEDUCTIONS= ";T2
390 DATA 3.15,40,3.85,40,4.15,40,3.75,35
400 DATA -999.99,-999.99
410 END
```

Notice in statement 110 that the REMARK doesn't have to contain any comments — it can be used to just create visual organization to the program by providing spacings in the listings. The main value of the REMARK statement, however, lies in the fact that they can make a program readable and, hence, more readily documented and usable by others.

An output sample from the above program:



```
PAY RATE= 3.15     HOURS= 40
GROSS= 126         NET= 99.54          DEDUCTIONS= 26.46

PAY RATE= 3.85     HOURS= 40
GROSS= 154         NET= 121.66         DEDUCTIONS= 32.34

PAY RATE= 4.15     HOURS= 40
GROSS= 166         NET= 131.14         DEDUCTIONS= 34.86

PAY RATE= 3.75     HOURS= 35
GROSS= 131.25      NET= 103.6875       DEDUCTIONS= 27.5625

NUMBER OF EMPLOYEES PROCESSED= 4
TOTAL NET= 456.0275
TOTAL DEDUCTIONS= 121.2225
```

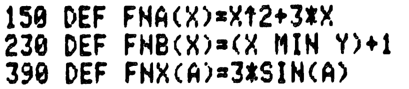
USER-DEFINED FUNCTIONS

The DEF FN Statement

In addition to the various built-in functions (like SIN, COS, SQR, etc.) provided by the system, the capability also exists which allows you to define your own functions and then use them, similar to the way the built-in functions are used. The facility to define functions is given by the DEF FN– (for define function) statement, with the following syntax form:

Line number DEF FN letter (numeric variable) = numeric expression

As you can see from the syntax form, a total of 26 functions can be defined (FNA through FNZ). Some examples are as follows:



```
150 DEF FNA(X)=X↑2+3*X
230 DEF FNB(X)=(X MIN Y)+1
390 DEF FNX(A)=3*SIN(A)
```

The "DEF" indicates that the line contains a user-defined function, and the FNA, FNB, etc., is the name of the function. The numeric variable enclosed within parentheses is the dummy variable of the function. The dummy variable is used only in the numeric expression to the right of the equal sign; that is, execution of the DEF statement does not result in any value being assigned to the dummy variable. The numeric expression on the right of the equal sign can be any valid combination of numeric variables, operators, and numeric functions that reduces to a single numeric value.

User-defined functions are utilized by specifying a value for the dummy variable, and then "calling" the function. To illustrate, suppose you are writing a program that needs to evaluate the following function:

$$f(x) = X^3 - 2X^2 + X$$

Let's say that at one point in the program, the function needs to be evaluated for X = –1, and at a later point in the program the function must be evaluated for X = 2. The function can be defined in the program as follows:

```
          .
          .
150 DEF FNA(X) = X↑3–2*X↑2+X
          .
          .
          .
```

Now, to evaluate the function for X = −1:

```
        .
        .
        .
240 PRINT FNA(-1)
        .
        .
        .
```

Line 240 effects a call to the function. When the BASIC interpreter encounters a line containing "FN", it looks for the corresponding DEF statement elsewhere in the program defining the function. In this case, the function is defined in statement 150. The call for the function in line 240 says to evaluate the function for a value of −1. The dummy variable (X) in line 150 receives the value of −1, and this value is then substituted for X wherever it appears in the numeric expression. Thus,

$$f(-1) = (-1)^3 - 2(-1)^2 + (-1) = -4$$

and statement 240 prints the value −4. The function is similarly evaluated for the value of 2:

```
        .
        .
        .
350    Y =2
360 PRINT   FNA(Y)
        .
        .
        .
```

In this case, the function is evaluated for the current value of Y. The dummy variable in the DEF statement receives the value of 2 this time.

$$f(2) = (2)^3 - 2(2)^2 + (2) = 2$$

The PRINT in statement 360 results in a value of 2. Notice, in this particular example, that the function is evaluated twice, but the actual formula defining the function needs to appear only once in the program. This gives you a considerable advantage in cases where long bulky formulae are involved, because you can utilize a user-defined function rather than having to repeat the formula each time it is to be evaluated.

The dummy variable specified in a DEF statement has an additional property that merits consideration: it is a *local variable*. That is, the dummy variable is not used anywhere outside the function. This is illustrated in the following brief program:

```
100 DEF FNA(X)=SQR(X)
110 X=10
120 Y=4
130 PRINT "FUNCTION EVALUATES TO ";FNA(Y)
140 PRINT "THE VALUE OF X IS ";X
150 END

RUN
FUNCTION EVALUATES TO 2
THE VALUE OF X IS 10
```

A simple function is defined in line 100, using X as the dummy variable. Line 110 sets the numeric variable X to 10; X won't be used for anything other than to verify that the local variable X in the function has no effect on the assigned value of the numeric variable X. In statement 120, Y is set to 4; this value is used in the function call in statement 130. When statement 130 is executed, the current value of Y (which is 4) is assigned to the dummy variable X in line 100, and the function is evaluated, finding the square root of 4. When line 140 is executed (and this is the important point being illustrated), X is still found to be equal to 10, even though the dummy variable X in line 100 received the value of 4.

A few remaining points should be made about user-defined functions. First, DEF FN — statements may appear anywhere in the program, either before or after they are referred to. Second, DEF FN — statements are not executed in the conventional sense; they only supply the appropriate formula when the function is called from some other point in the program. Third, one function can call another function:

```
250 DEF FNA(X)=X↑3+2*X
260 DEF FNB(X)=FNA(X)*0.175
```

In evaluating FNB(X), the BASIC interpreter first computes the value of FNA(X), and then uses that value to find FNB(X).

The following program employs a function within a function to output a table of centimeters, inches, feet, and yards:

```
100 DEF FNA(C)=C*0.3937
110 DEF FNB(C)=FNA(C)*0.08333
120 PRINT "CENTIMETERS","INCHES","FEET","YARDS"
130 FOR C=1 TO 10
140 PRINT C,FNA(C),FNB(C),FNB(C)/3
150 NEXT C
160 END
```

```
RUN
CENTIMETERS       INCHES            FEET              YARDS
 1                 0.3937            0.032807021       0.0109356736667
 2                 0.7874            0.065614042       0.0218713473333
 3                 1.1811            0.098421063       0.032807021
 4                 1.5748            0.131228084       0.0437426946667
 5                 1.9685            0.164035105       0.0546783683333
 6                 2.3622            0.196842126       0.065614042
 7                 2.7559            0.229649147       0.0765497156667
 8                 3.1496            0.262456168       0.0874853893333
 9                 3.5433            0.295263189       0.098421063
 10                3.937             0.32807021        0.109356736667
```

SUMMARY

There are 5 arithmetic operators: ↑ (exponentiation), * (multiplication), / (division), + (addition), – (subtraction). Exponentiation is performed first, then multiplication and division, then addition and subtraction. Relational operators are formed with the symbols < , > , and = to make comparisons like "less than", "greater than", etc. Two other relational operators are the MIN/MAX operators which select the smallest or largest operand. The logical operators are AND, OR, and NOT, which return a 1 or 0. Assignment statements employ the assignment operator, or equals sign, to assign a value to a variable.

You can input data to the Graphic System with READ and DATA statement combinations. RESTORE resets the DATA statement "pointer" to the first data item in the DATA statement. Data can also be input with the INPUT statement, which enables the Graphic System to receive input from the keyboard while a program is running.

The output of information from the Graphic System is accomplished with the PRINT statement. The use of commas with the PRINT statement results in the generation of automatic tabs and 18 character wide print fields. Semicolons suppress this tab. You can output string constants with a PRINT statement, enclosing the strings within quotation marks.

Terminating a program is done with the END statement. Another keyword, STOP, allows you to re-start a program because execution of STOP results in a printed indication (on the screen) of what point in the program the STOP occurred.

Unconditional transfers occur with a GOTO statement; control is always (unconditionally) passed to the specified statement numbers. GOTO statements can be used to form loops. Conditional transfers are initiated by IF . . . THEN . . . combinations. If the condition is true, then control branches to the specified statement number. If the condition is not true, control "falls through" to the next statement in the program.

FOR/NEXT loops are the most convenient types of loops to set up. You can branch out of a FOR/NEXT loop, but you should not branch into the body of a FOR/NEXT loop. FOR/NEXT loops can be "nested", i.e., one inside the other.

The "computed GOTO" capability is given by the GOTO . . . OF . . . statement. This statement allows multiple branching of the flow of execution.

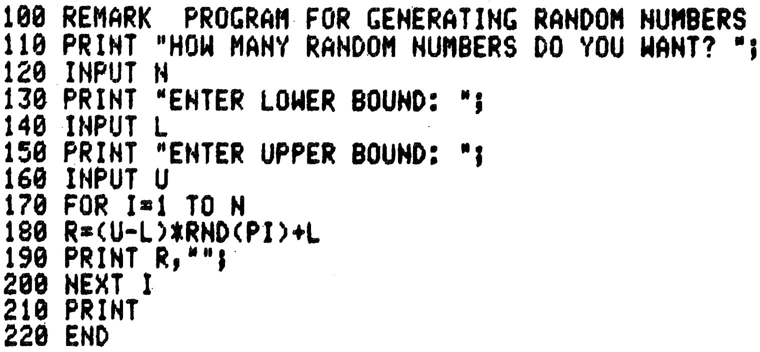
Documenting a program can be accomplished with a REMARK statement. This increases the readability of a program listing so you can readily determine what the program is doing at various points.

EXAMPLE PROGRAMS

TITLE: Random Number Generator

DESCRIPTION: This program generates random numbers between upper and lower bounds of your choosing.

PROGRAM LISTING:



```
100 REMARK  PROGRAM FOR GENERATING RANDOM NUMBERS
110 PRINT "HOW MANY RANDOM NUMBERS DO YOU WANT? ";
120 INPUT N
130 PRINT "ENTER LOWER BOUND: ";
140 INPUT L
150 PRINT "ENTER UPPER BOUND: ";
160 INPUT U
170 FOR I=1 TO N
180 R=(U-L)*RND(PI)+L
190 PRINT R,"";
200 NEXT I
210 PRINT
220 END
```

METHODOLOGY: A FOR/NEXT loop generates the random numbers. The value of π is used as a convenient "seed" value for the RND function in line 180. Since a PRINT statement can not end with a comma, line 190 ends with a null character string and a semicolon. The null means "print nothing"; this results in the 4 column tabbed output.

OPERATING PROCEDURE: Type RUN and press RETURN. The program responds with "HOW MANY RANDOM NUMBERS DO YOU WANT"? Enter the desired number and press RETURN. The next two promptings are "ENTER LOWER BOUND" and "ENTER UPPER BOUND". Enter the appropriate bounds and the random numbers are output. If you want random numbers between 1 and 10, then the lower bound is 1, and the upper bound is 10.

OUTPUT SAMPLE:

```
RUN
HOW MANY RANDOM NUMBERS DO YOU WANT? 100
ENTER LOWER BOUND: 1
ENTER UPPER BOUND: 10
 8.06253023309      3.49452009355      3.47249467269      5.65230329655
 1.46031246408      9.42154916125      1.91884963647      8.44034465532
 9.43662671277      2.64332217721      9.56418734915      6.79346010227
 4.91204290933      9.2955484389       8.38059465386      1.15883941292
 4.50283303307      5.92623958907      8.0108819673       4.76276783949
 2.37431571318      1.81424598302      3.16017263441      9.07579053022
 9.42104963425      1.94540973072      7.11631586147      1.1044556253
 1.18289807808      1.12070632405      2.30400279756      2.88272772721
 4.77386107842      2.72789092402      5.96455334707      3.54093944419
 6.28363135775      6.48186228278      4.52631798054      4.2799930657
 7.59835670447      7.62061315861      8.19653541756      6.25859339877
 7.05286598517      1.76215720037      5.15883483384      1.53424262303
 7.13491981927      4.44961488065      8.08340144449      1.44471496072
 4.66366132363      3.34061139323      9.21866818802      8.77340925589
 6.19764851123      6.47469375252      2.09128502923      1.2445074642
 4.31842915172      5.95751888745      2.93958236498      2.81490282133
 2.99685242214      3.04950556315      4.77681740476      8.27955933389
 4.28198074348      4.79338673746      8.94370345726      1.23298047232
 1.54485106315      3.15105171868      3.87210303585      1.71695578975
 7.63721875426      5.77413548362      6.43784784886      3.15637589492
 8.45208051091      3.07140199311      9.81246331627      7.60081502094
 8.04899637353      1.66298555337      3.40065350086      5.94743260428
 8.97129656976      8.56840546867      2.72977551132      7.75569707074
 9.55994745551      9.94086066071      3.68373177545      9.44914266669
 7.57344373591      8.65104884559      5.07876534413      8.04398992495
```

TITLE: **Invested Value of a Dollar**

DESCRIPTION: The program outputs a table showing the value of a dollar for years 1 through 25 at 5, 6, and 7 percent compound interest (compounded annually).

PROGRAM LISTING:

```
100 REMARK INVESTED VALUE OF A DOLLAR
110 PRINT
120 PRINT "                    VALUE OF A DOLLAR"
130 PRINT
140 PRINT " YEAR"," 5 PERCENT"," 6 PERCENT"," 7 PERCENT"
150 PRINT " ----"," ---------"," ---------"," ---------"
160 FOR N=1 TO 25
170 PRINT N,"";
180 FOR R=0.05 TO 0.07 STEP 0.01
190 PRINT (1+R)↑N,"";
200 NEXT R
210 PRINT
220 NEXT N
230 END
```

METHODOLOGY: The computations are done in a nested FOR/NEXT loop. Lines 170 and 190 utilize the "null string" technique to achieve tabbed output and avoid the problem of PRINT statements not being able to end with a comma.

OPERATING PROCEDURE: Type RUN and press RETURN. No data is required.

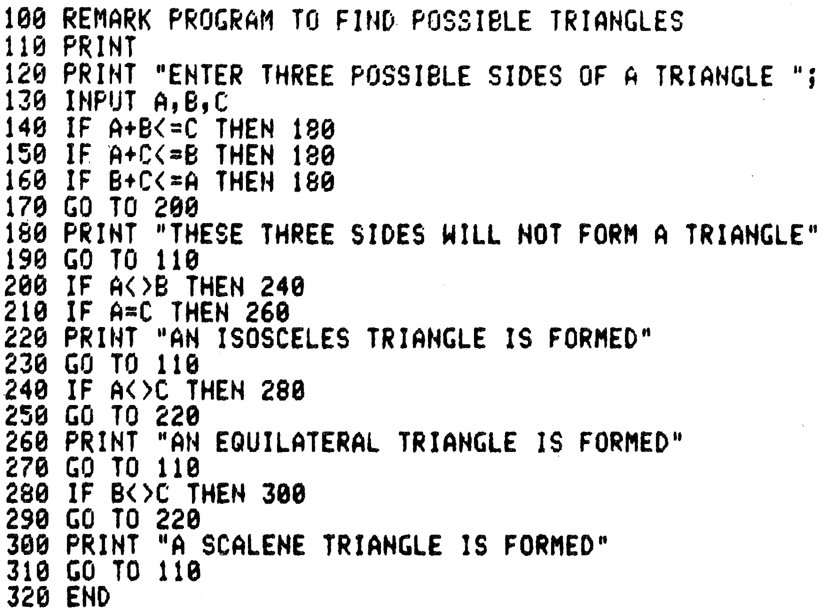
OUTPUT SAMPLE:

```
RUN

                    VALUE OF A DOLLAR

 YEAR              5 PERCENT         6 PERCENT         7 PERCENT
 ----              ---------         ---------         ---------
 1                 1.05              1.06              1.07
 2                 1.1025            1.1236            1.1449
 3                 1.157625          1.191016          1.225043
 4                 1.21550625        1.26247696        1.31079601
 5                 1.2762815625      1.3382255776      1.4025517307
 6                 1.34009564063     1.41851911226     1.50073035185
 7                 1.40710042266     1.50363025899     1.60578147648
 8                 1.47745544379     1.59384807453     1.71818617983
 9                 1.55132821598     1.689478959       1.83845921242
 10                1.62889462678     1.79084769654     1.96715135729
 11                1.71033935812     1.89829855834     2.1048519523
 12                1.79585632602     2.01219647184     2.25219158896
 13                1.88564914232     2.13292826015     2.40984500019
 14                1.97993159944     2.26090395575     2.5785341502
 15                2.07892817941     2.3965581931      2.75903154072
 16                2.18287458838     2.54035168469     2.95216374857
 17                2.2920183178      2.69277278577     3.15881521096
 18                2.40661923369     2.85433915291     3.37993227573
 19                2.52695019538     3.02559950209     3.61652753503
 20                2.65329770514     3.20713547221     3.86968446249
 21                2.7859625904      3.39956360055     4.14056237486
 22                2.92526071992     3.60353741658     4.4304017411
 23                3.07152375592     3.81974966157     4.74052986298
 24                3.22509994371     4.04893464127     5.07236695338
 25                3.3863549409      4.29187071974     5.42743264012
```

TITLE: **Program to Find Possible Triangles**

DESCRIPTION: This program inputs three numbers which are possible sides of a triangle. The three numbers are examined; if they can be configured into a triangle, the program tells you what kind of triangle is formed. Otherwise, the program informs you that the three numbers can not be sides of a triangle.

PROGRAM LISTING:



```
100 REMARK PROGRAM TO FIND POSSIBLE TRIANGLES
110 PRINT
120 PRINT "ENTER THREE POSSIBLE SIDES OF A TRIANGLE ";
130 INPUT A,B,C
140 IF A+B<=C THEN 180
150 IF A+C<=B THEN 180
160 IF B+C<=A THEN 180
170 GO TO 200
180 PRINT "THESE THREE SIDES WILL NOT FORM A TRIANGLE"
190 GO TO 110
200 IF A<>B THEN 240
210 IF A=C THEN 260
220 PRINT "AN ISOSCELES TRIANGLE IS FORMED"
230 GO TO 110
240 IF A<>C THEN 280
250 GO TO 220
260 PRINT "AN EQUILATERAL TRIANGLE IS FORMED"
270 GO TO 110
280 IF B<>C THEN 300
290 GO TO 220
300 PRINT "A SCALENE TRIANGLE IS FORMED"
310 GO TO 110
320 END
```

METHODOLOGY: IF . . . THEN . . . statements are used to examine the three sides and to determine into which one of four categories (isosceles, equilateral, scalene, or none) the sides fall.

OPERATING PROCEDURE: Type RUN and the program prompts with "ENTER THREE POSSIBLE SIDES OF A TRIANGLE". Enter the three possible sides, separated with commas, and press RETURN. Terminate the program by pressing BREAK BREAK.

OUTPUT SAMPLE:

```
RUN

ENTER THREE POSSIBLE SIDES OF A TRIANGLE 3,4,5
A SCALENE TRIANGLE IS FORMED

ENTER THREE POSSIBLE SIDES OF A TRIANGLE 5,5,4
AN ISOSCELES TRIANGLE IS FORMED

ENTER THREE POSSIBLE SIDES OF A TRIANGLE 6,6,6
AN EQUILATERAL TRIANGLE IS FORMED

ENTER THREE POSSIBLE SIDES OF A TRIANGLE 5,4,11
THESE THREE SIDES WILL NOT FORM A TRIANGLE

ENTER THREE POSSIBLE SIDES OF A TRIANGLE
```

Section 3

DIRECTIVES

PROLOGUE

At this point, you are familiar with the essentials of programming in BASIC. You know how to input data and output results, you can set up loops using several methods, you know about conditional transfers, unconditional transfers, and computed GOTO's. Now, before progressing further into some of the more advanced aspects of the Graphic System BASIC language, it is appropriate to become familiar with the statements in the language that give you control over the operation of the system. This subject is presented here, and is also discussed in detail in the accompanying system reference manual. For the beginning programmer these statements (called "directives") and the associated concepts are best understood in the context of programming, because they are intended to be aids in the development of programs.

Directives are statements in the BASIC interpreter's repertoire which are primarily intended to give you control over the operation of the machine, but also can be part of a program. You have already been briefly introduced to one of these directives—the RUN statement. RUN is a program control directive which tells the BASIC interpreter to begin execution of the program currently in memory. Other program control directives enable you to obtain a listing of the program which is currently in memory, remove lines of code from a program, resequence program statement numbers, etc. There are also directives which permit you to specify the parameters of operating "environment" of the Graphic System. This includes things like establishing the desired trigonometric modes (degrees, radians, grads). The operating environment also includes the comparison limits to be used in relational (IF . . . THEN . . .) statements. That is, if you set X equal to 1/3, you would expect (3 * X) to be equal to 1, but this is not the case. This is because 1/3 is represented as .333333333333 in machine language, and therefore (3 * X) is .999999999999, not 1. To overcome this phenomenon, you can specify the "degree of accuracy" to be used. Typically, however, you need not be concerned with this sort of thing, because there is a built-in default "closeness value" parameter which is used.

In addition to the directives that give you program and environmental control, there are directives which give you control over the operation of the internal magnetic tape unit. This is especially useful because it allows you to store programs or data on tape for later use. Without this facility, you would have to enter a program from the Graphic System keyboard every time you want to use it.

One further group of directives facilitates memory monitoring. With these directives, you can determine the amount of memory a program requires, and you can also determine the amount of free space remaining in the memory. These memory monitoring directives let you conveniently determine the amount of tape storage that is required to store a program. They also let you determine if sufficient memory remains to append, or add, a program from tape to a program that is already in memory.

PROGRAM CONTROL

The RUN Statement

The program control statement that you are already somewhat familiar with is RUN. The general form for this statement is:

```
[Line number] RUN [Line number]
```

Note that there is an optional statement number preceding RUN, indicating that this statement can be used in a program. Normally, this is not done. The effect is to transfer control to the statement number following the word RUN, or, in the absence of any such number, control is transferred to the first statement in the program.

If you type RUN and do not follow it with a statement number, the Graphic System begins program execution starting at the lowest numbered statement in memory. RUN followed by a statement number starts program execution at the specified statement number. This is useful when you want to bypass a portion of your program. This might be the case when you want to examine the results of some part of the program which is near the end, without running the whole program.

Entering a statement such as RUN 500 causes execution to begin at the specified statement number, and all previous statements are skipped. This means that all variables which are defined earlier in the program are either going to have no values assigned to them, or they are going to contain their previous values (if any). This can affect your results.

The LIST Statement

To obtain a listing of your program, just type LIST. The syntax form for this statement is:

```
[Line number] LIST [line number [, line number] ]
```

As with RUN, you can include LIST as part of your program. Generally, however, this statement is used without a preceding statement number (i.e., not as part of a program) for the purpose of obtaining a listing of the statements in a program. LIST with no parameters causes the entire program to be listed. LIST with one line number causes the specified statement to be listed. If you specify a beginning line number and an ending line number, then only the program statements bounded by the specified statement numbers are listed.

Examples:

LIST	Lists entire program on the screen.
LIST 500	Lists statement 500.
LIST 100,400	Lists the portion of the program beginning at statement 100 and continuing through 400.

If you want to list everything between the end of a program and a specified line number in the program, and you don't know the number of the last statement, you can just specify an artificially high number as the ending line number. For instance, LIST 250,9999 lists everything between 250 and the end of the program, even though the last line number might be 550 or 600.

The DELETE Statement

Frequently you will want to delete lines from your program. For example, a common program debugging technique is to place extraneous PRINT statements at key positions in the program so that you can examine the contents of variables. Later, you will probably want to remove these statements from the program. This is facilitated by the DELETE statement, formed as follows:

```
                       ⎧ ALL                          ⎫
[line number] DELETE   ⎨ line number [, line number]  ⎬
                       ⎩ variable list                ⎭
```

DELETE ALL does what you would expect: it deletes the entire program in memory, plus all variables. In effect, it "erases" memory so that you can start all over. DELETE followed by a line number results in the deletion of the specified line from memory. DELETE followed by two line numbers deletes all lines in memory between and including the specified line numbers. If you want to delete a variable from memory, just type DELETE followed by the variable (or variables). You can delete several variables with DELETE by separating them with commas. Once you delete a variable, it is gone. It isn't set to zero, it is removed from memory completely.

The RENUMBER Statement

While you are in the process of developing a program, you frequently make deletions and additions. This can result in a disorganized procession of statement numbers. Also, you can end up making so many additions between two originally adjacent statements that no room exists for further additions. For whatever reason, you might want to renumber the statements in your program. This can be done with the following statement:

```
[Line number] RENUMBER [numeric expression[, numeric expression[, line number]]]
```

This statement can be used with no parameters, or 1, 2, or 3 parameters. When RENUMBER is used by itself, the renumbering process makes the first statement with line number 100 or greater become line number 100 unless line numbers below 100 are part of the program. (Line numbers below 100 remain unchanged.) Subsequent statements are numbered with an increment of 10 (i.e., the first statement is 100, the second is 110, and the third is 120, etc). If you don't want the first statement to be 100, you can indicate your preference with the first parameter. For instance, to renumber a program so that the first statement is 500, just say RENUMBER 500. This results in the first number statement in the program being set to 500. The second statement is 510, the third 520, etc. (The default numbering increment of 10 is still in effect.)

If the default increment of 10 doesn't suit your needs, this can be altered through use of the second parameter in RENUMBER. This way, a statement like RENUMBER 100,5 causes the first statement in the program to be 100, the second 105, followed by 110, etc. Notice that in order to use the second parameter, you must specify some number for the first parameter. This only makes sense—if you attempt to obtain a numbering increment of 5 by saying RENUMBER 5, you are presented with a program numbered 5, 15, 25 and so on. The third parameter of RENUMBER allows you to renumber everything starting with the specified statement. It might happen that you are developing a program with statement numbers like the following:

```
100
110
120
 .
 .
 .
290
300
310
```

Suddenly, you discover that you need to add fifteen or twenty statements between statement numbers 290 and 300. No problem. You can easily obtain the necessary room through use of the third parameter of RENUMBER by specifying something like the following:

```
RENUMBER 500, 10, 300
```

This changes the statement numbers of the program as indicated below:

Before	**After**
100	100
110	110
120	120
.	.
.	.
.	.
.	.
290	290
300	500
310	510

The RENUMBER 500, 10, 300 statement instructs the BASIC interpreter to renumber the program, using 500 as the statement number and an increment of 10, beginning at statement number 300. It is now convenient to add numerous additional statements after statement 290. Previously, you could add only 9 statements after 290 before you would have started writing over, and thereby losing, statement 300.

RENUMBER used with 0, 1, or 2 parameters does not renumber existing statements with line numbers less than 100. That is, if you have a program with line numbers 2, 4, 6, and 8 and type RENUMBER 100, nothing happens. However, RENUMBER 100, 10, 2 changes the 2, 4, 6, 8 to 100, 110, 120, 130.

By now, you may have wondered what happens when you RENUMBER a program containing GOTO statements. After all, if you have a statement like GOTO 150 or GOTO 3 of 115, 175, 205 and then renumber everything, you would think that control is likely to end up "going to" the wrong place. Fortunately, that potential problem is averted by the fact that the renumbering process updates all GOTO type statements. That is, if you have a statement like GOTO 355, and then do a RENUMBER so that statement number 355 becomes, for instance, statement number 450, the GOTO statement will automatically be updated to GOTO 450. However, if like numbers are used as arguments in CALL statements, RENUMBER will not update them.

Now, to quickly review the RENUMBER statement:

* RENUMBER used by itself with no parameters resequences the program so that the first statement containing line number 100 or greater becomes line number 100, and the increment is 10, as in 110, 120, 130, 140, etc. The default parameters of RENUMBER are 100, 10, 100.

* RENUMBER with one parameter sets the number of the first statement in the program to the specified number. The increment remains at 10.

* When used with two parameters, RENUMBER sets the number of the first statement in the program to the number specified in the first parameter. The increment that is used to number subsequent statements in the program is the number specified in the second parameter.

* RENUMBER with three parameters is similar to the two parameter version, except that the numbering begins with the statement number specified in the third parameter.

* The renumbering process automatically updates all statement number references (like GOTO 510) in the program, except those that appear in CALL statements (like CALL "BAPPEN", 2000).

MEMORY MONITORING

Memory Bytes

One very useful piece of information to have about any given program is its size, i.e., the amount of memory required by the program. The size of a program is determined not only by the number of statements involved, but also by the usage of variables and the amount of data that is stored. Obviously, the more variables and data you use in a program, the more demand you make on the Graphic System memory.

When you begin to store programs on magnetic tape, you need to know the approximate size of the programs you intend to store so that you can set aside an appropriate amount of space on the tape. Also, one clue to the efficiency of a program is the amount of memory it requires. That is, given two programs that perform an identical task, the one that requires the least memory is likely to be the most efficient.

The unit used to describe the memory requirements of a program is the *byte*. A byte is a group of eight consecutive binary digits (1's and 0's) which are operated upon as a unit. A statement in a program which we see represented as alphanumeric text might be represented in internal binary notation by 20 or 30 bytes.

The SPACE Function

There is a directive in the language which you can use to readily determine the amount of memory, expressed in bytes, that a program requires. This directive is actually a function like PI; it is the SPACE function. If you type SPACE and press RETURN, a number appears on the display which indicates the upper bound of the number of bytes required to store the program currently in memory. The number returned by SPACE corresponds to the number of lines in the program multiplied by 72. (72 is the maximum number of characters that can be on one line of the screen.) SPACE can also appear in a program, as in the following lines:

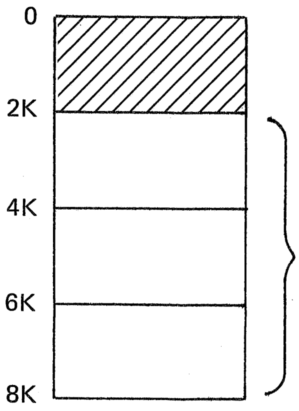
```
155 PRINT SPACE
180 LET X=SPACE * 2
```

The MEMORY Function

A second memory monitoring directive is the MEMORY function. This function provides the amount of "free" (unused) memory, expressed in bytes, that remains in the random access read/write memory. This function is similar to the SPACE function in the way it is used: just type MEMORY to obtain a tally of the amount of free memory remaining. Or, like SPACE, it can be used in a program:

```
230 PRINT MEMORY
410 LET X=MEMORY + 50
```

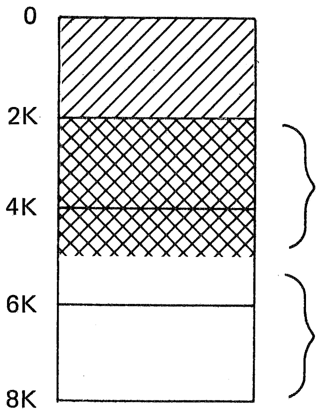
It is appropriate at this point in the discussion to give the structure of the random access read/write memory a cursory examination. If you have a system that has, say, 8K of memory, this means 8 kilobytes or 8000 bytes. The 8K is a "ball park" figure, because an 8K machine actually contains 8192 bytes. (In binary devices, nearly everything depends on powers of two. 8192 is 2^{13}.) When you first apply power to an "8K" memory, the memory appears as follows:



First 2K of memory (approximately) reserved for the processor as a working area.

Remaining 6K is free memory. The SPACE function returns a 0 because there is no program in memory. The MEMORY function returns a value of about 6000 because that is how much free memory exists.

Now, let's assume you have written a program that requires 3000 bytes. The memory now looks like the following conceptualization:



First 2K still reserved as a working area for the processor.

3K utilized to contain the program. The SPACE function will now return a value of 3000.

3K of free memory. The MEMORY function will return a value of approximately 3000, indicating the amount of memory remaining.

Now that you are armed with these two memory monitoring directives, you are ready to tackle the task of using the internal magnetic tape unit.

PROGRAMS ON TAPE

The Graphic System has the ability to store programs on magnetic tape so that you do not have to enter a program from the keyboard every time you want to run it. You can also store data on the tape, but the discussion of this subject is deferred until the Extended I/O section.

The operation of the internal magnetic tape unit involves another group of directives. These directives give you the necessary control over operation of the tape so that you can do the following things:

* Create files on the tape. Files are sections of tape that can contain programs. Before you can store a program on tape, you must create a file.

* Store programs on tape in the files that you have set up.

* Retrieve programs from the tape. The way you retrieve a program is by telling the BASIC interpreter to load the contents of a specified file into memory.

* Stack, or append, programs in memory, one after another.

* List out, on the display, information about the contents of the tape.

Near the beginning of the tape there is a pattern of small holes. Inside the tape drive mechanism, there is an optical sensing device which contains a light source and a light detector. The tape is routed between the light source and the light detector. The pattern of holes in the tape allows light to pass from the source to the detector. As a result, the tape drive mechanism is able to locate the beginning (and the end) of the tape by the action of light activating the detector when the holes pass between the two.

The FIND Statement

Assuming that you have a program to store on tape, the first thing you have to do is position the beginning of the desired file at the recording head. This is done with the positioning directive:

```
[Line number] FIND numeric expression
```

The *numeric expression* refers to the file number you are requesting. To position the read/write head at the beginning of the tape (rewind), type FIND 0. This instructs the tape drive to position the tape at the load point, which is the beginning of the tape. (You can also do this by pressing the REWIND key. The only difference between the two methods is that one is programmable, the other is not.)

File 0 does not actually exist. You can not store anything in file 0; it is used only to indicate the beginning of the tape. However, once you have "found" file 0, you are then able to create a file that can store a program.

The MARK Statement

Files are created with the MARK statement:

```
[Line number] MARK numeric expression, numeric expression
```

The first parameter refers to the quantity of files you want to set up, and the second indicates the length in bytes of the file or files being created.

If you want to create one file with a length of 1000 bytes, enter

```
MARK 1,1000
```

Similarly, if you want to create 5 files with a length of 1200 bytes, enter

```
MARK 5,1200
```

Typically, however, you only create files one at a time as they are needed.

It is a good idea to set up files that are larger than necessary; this way room remains in the file to accommodate any later additions to the program being stored. You determine the maximum amount of storage that a given program requires through the use of the SPACE statement discussed earlier. Also, a good "rule of thumb" to approximate the storage requirements for a program is to figure 40 bytes per line of code. (This is a rough approximation—some lines take considerably more, and some lines take considerably less.)

Now, suppose you have written a program and it is currently residing in the random access memory. You want to store it on tape. You have determined that the program requires, say, 1500 bytes of storage, and you are working with a blank tape. Enter:

```
FIND 0
```

and the beginning of the tape is positioned at load point. Now, to create the file, type

```
MARK 1,3000
```

and the Graphic System sets up one file with sufficient length to contain the program twice over. (The 3000 was chosen arbitrarily.) This file now exists with a *physical* length of 3000. It currently has a *logical* length of 0 because, as yet, nothing has been stored there. Before the program can be stored in this newly created file, the tape must be positioned at the beginning of the file. To do this, just type

```
FIND 1
```

and the tape drive positions the tape at the beginning of the file.

The SAVE Statement

The actual storing, or "saving", operation is initiated by another directive:

```
[Line number] SAVE [line number[, line number]]
```

SAVE used by itself causes the entire program currently in memory to be stored on tape in ASCII code. SAVE with one line number stores only the specified line number. SAVE with two line numbers stores only the part of the program bounded by and including the two specified line numbers.

As soon as you type SAVE and press RETURN, the system records a copy of the program in memory on the tape. (It will record those statements which are preceded by a line number.) The file now has, in addition to its *physical* length, a *logical* length. In this case, the physical length is 3000 bytes as established by the MARK statement, and the logical length is 1500 bytes, because the program saved was 1500 bytes.

Before going any further, let's review what has happened thus far:

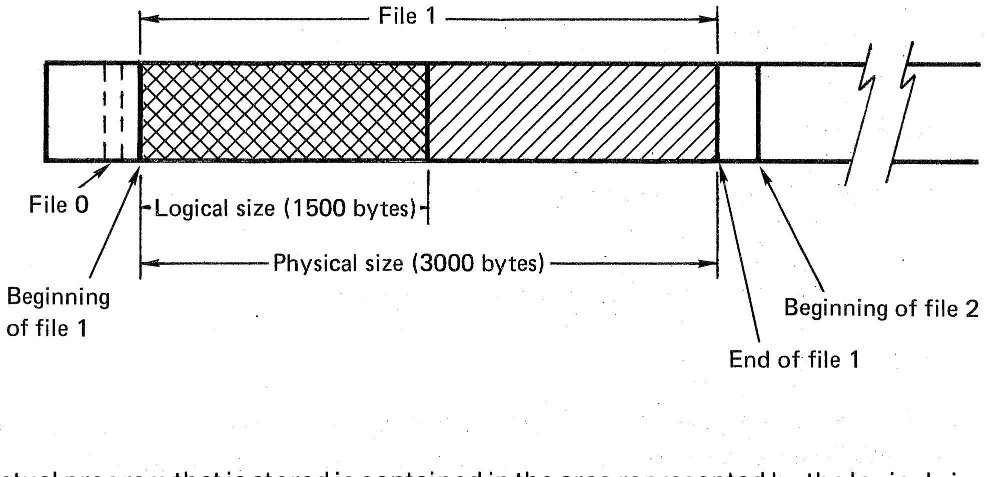
* FIND 0 positioned the tape at the beginning load point.

* MARK 1,3000 created a file with a physical length of 3000 bytes.

* FIND 1 positioned the tape at the beginning of the file.

* SAVE caused the machine to make a copy of the program in memory on the tape in file 1.

Actually, the MARK instruction not only "marks" the beginning and end of the file that is to contain the program; it also "marks" the place where the following file will be located. (The "marking" is accomplished magnetically.) This is done so that the machine can "find" the correct place to "mark" the next file. Following the execution of FIND 1 and SAVE, the tape is conceptually like this:



The actual program that is stored is contained in the area represented by the logical size. Note that the logical size is less than the physical size, thereby permitting expansion of the program at a later date.

The BSAVE Routine

Just like you can store programs in ASCII code with the SAVE command, you can store programs in binary code with the BSAVE routine. It's called a routine because it requires a CALL command:

```
            [Line Number]  CALL "BSAVE"
```

NOTE

A 4051 Graphic System must be equipped with a Binary Loader ROM Pack before using this routine.

Programs are stored in the Graphic System's memory as binary code. When a program is stored on tape with the SAVE command, it must first be translated to ASCII. When the BSAVE routine is used, the program doesn't need to be translated, so the BSAVE routine takes less time. On the other hand, ASCII code takes up less tape storage space than binary code.

The BSAVE routine sends a copy of the current program to the tape 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. Unlike SAVE, line numbers cannot be used with BSAVE.

To execute the BSAVE routine, the read/write head of the magnetic tape unit must be positioned at the beginning of a file marked BINARY 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, you can use one of the following methods:

METHOD 1 Allocate all of the space the program uses in memory. For example, if your system has 32K of memory:

```
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.

METHOD 2 Allocate enough space to hold the entire contents of memory. For example:

```
MARK 1,32000
```

This method may waste a lot of space on the tape if the current program is small.

NOTE

The actual storage capacity of memory is approximately 32K or 32000 bytes. Of this 32000 bytes, about 2000 are reserved by the processor for a work area. The MEM function only considers the 30000 bytes that are free. By using 32000 as a constant, you can make sure that you consider all of the possible storage capacity of memory.

The following example stores the current BASIC program in binary code.

```
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.

The OLD Statement

Now let's suppose that, after performing some unrelated operations, you want to run an ASCII program previously saved on file 1. Retrieving the program is facilitated with the following statement:

```
[Line number] OLD
```

To get the program back into memory, enter

```
FIND 1
OLD
```

FIND 1, again, positions the tape at the beginning of file 1. OLD causes the Graphic System to first delete everything in memory (as though a DELETE ALL statement is executed), and then copy the logical contents of the file into memory.

When you want to save another program on tape, the process is essentially the same. Assuming that you have already utilized file 1 as in the previous discussion, the first thing to do is to type

```
FIND 2
```

The machine is able to locate file 2 because the process of "marking" file 1 also marks the beginning of file 2. Now, supposing you want to establish one file with a length of 2000 bytes, you enter

```
MARK 1,2000
```

and the file is created. (Do not confuse the file number with the number of files specified by the MARK statement.) You can now store a program in the second file by entering

```
FIND 2
SAVE
```

and the system saves the program in memory on the tape, in file 2.

The BOLD Routine

```
[Line number] CALL "BOLD"  [,I/O address]
```

NOTE

A 4051 Graphic System must be equipped with a Binary Loader ROM Pack before using this routine.

Any program that is in binary code can be loaded into memory by using the BOLD routine. A program stored by the BSAVE routine is usually the program retrieved by the BOLD routine.

Loading a Binary Program

To retrieve a binary program from tape, the read/write head of the tape drive 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.

Using AUTO LOAD with Binary Files

The AUTO LOAD key finds the first file on the magnetic tape and then executes an OLD command automatically. The OLD command loads and executes the first program on file. Because the OLD command does not work with binary program files, the first program on tape must be an ASCII program if you use the AUTO LOAD key.

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

Suppose that file 2 is the binary program you wish to load with the AUTO LOAD key. Then file 1, an ASCII program file, should contain this program:

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

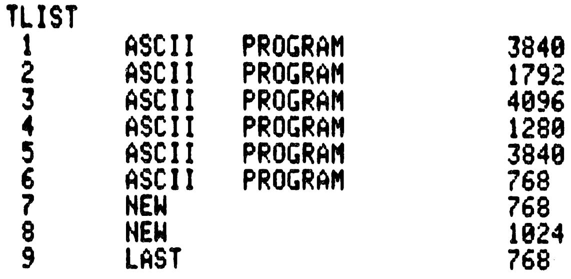
When the AUTO LOAD key is pressed, the program in file 1 is loaded and executed. This program in turn finds and loads the desired binary program in file 2. Since the CALL "BOLD" statement is executed under program control, the binary program in file 2 is executed after being loaded.

The TLIST Statement

At this point, it should be apparent that you need the facility to determine how many files exist on the tape, and the status of each file. This capability is provided by the additional directive:

```
[Line number] TLIST
```

The TLIST statement lists out (on the screen) information about the tape that is in the tape drive assembly. This listing is typified by the following:



```
TLIST
 1       ASCII   PROGRAM          3840
 2       ASCII   PROGRAM          1792
 3       ASCII   PROGRAM          4096
 4       ASCII   PROGRAM          1280
 5       ASCII   PROGRAM          3840
 6       ASCII   PROGRAM          768
 7       NEW                      768
 8       NEW                      1024
 9       LAST                     768
```

The numbers in the left-hand column indicate the number of the file. The center column identifies the type of file. In this case, the first six files contain programs. Files 7 and 8 are "NEW", indicating they are "empty" and available for use. The "LAST" file (file 9) is self-explanatory: it is the next one to be MARKed and used. The right-hand column of numbers indicates the physical size, in bytes, of each file, and is always a multiple of 256.

NOTE

If you "re-mark" a file, then the file following the one that is "re-marked" always becomes the "LAST" file. Any files that may have been beyond the "re-marked" file are lost.

The APPEND Statement

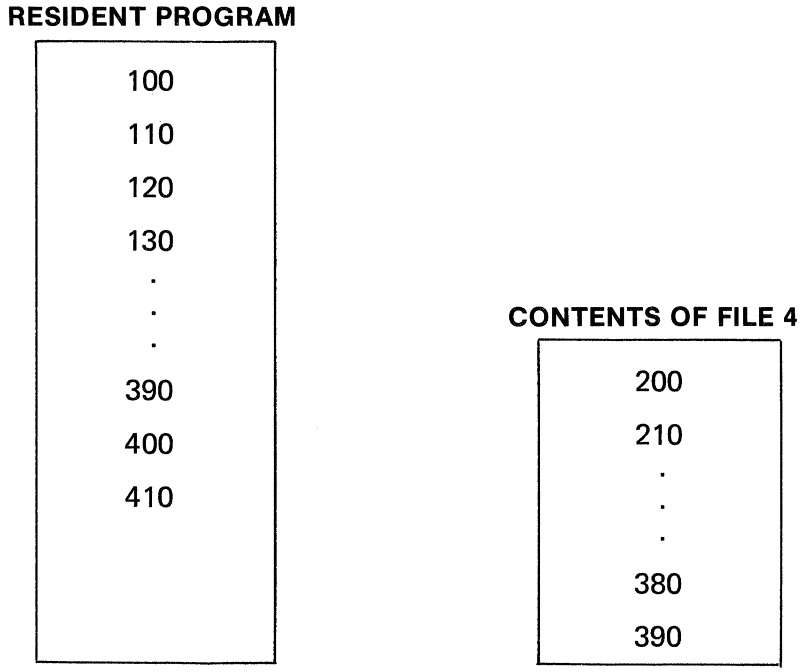
Three tape control directives enable you to append routines or programs which are on tape to the program which is resident (currently existing) in memory. The first statement that is used is APPEND, formed as follows:

```
[Line number] APPEND line number [, numeric expression]
```

The appended program is inserted into the resident program beginning at the specified line number. If you specify a numeric expression, the program lines are always renumbered using the specified increment beginning at the specified line number and continuing through to the end of the program. The file that is appended to the program in memory is the current file on the tape: that is, if you precede APPEND with FIND 3 (for example), then file 3 is the file that is appended. It is important, therefore, to make sure you precede APPEND with an appropriate FIND statement.

The APPEND statement requires that there must be a line number in the resident program which corresponds to the number specified after the word "APPEND". If you state APPEND 500, there must be a statement number 500 in the program currently in memory.

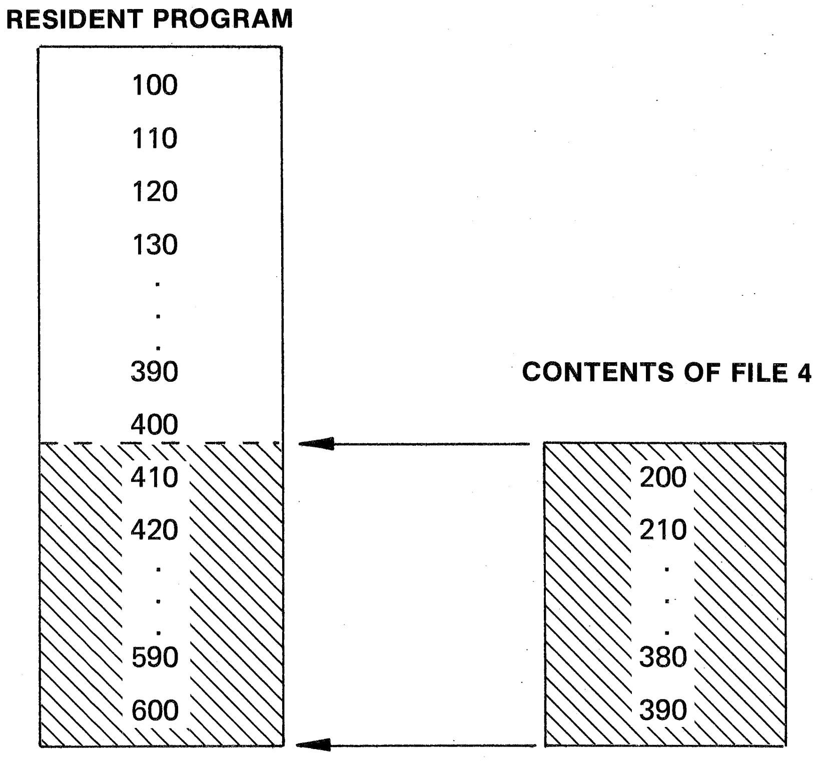
Typically, APPEND is used to "add-on" program statements to the end of the resident program. To illustrate, assume that you have a program which begins at statement 100, and ends with statement 410. You also have, on file 4 of the tape, additional statements which you want to place at the end of the resident program. The line numbers of the statements contained in file 4 can be almost anything—for example they run from 200 through 390. The situation is depicted in the following:



Now to append file 4 to the end of the resident program:

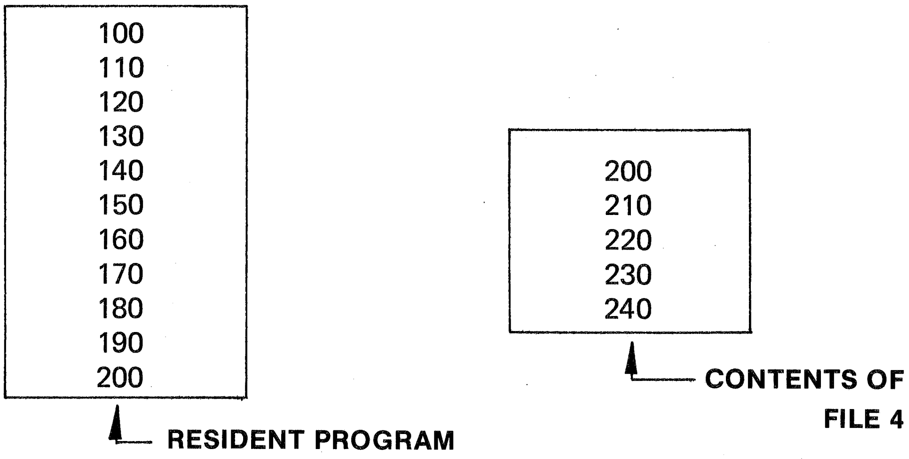
```
FIND 4
APPEND 410
```

The first statement of file 4, which was line number 200, becomes line number 410, and the old line number 410 is lost. (It could have been an END, STOP, REMARK, etc.) Following execution of APPEND, the resident program now looks like the following:



Notice that the contents of file 4 have been added on to the end of the resident program. Also, the appended portion has been renumbered with a default increment of 10.

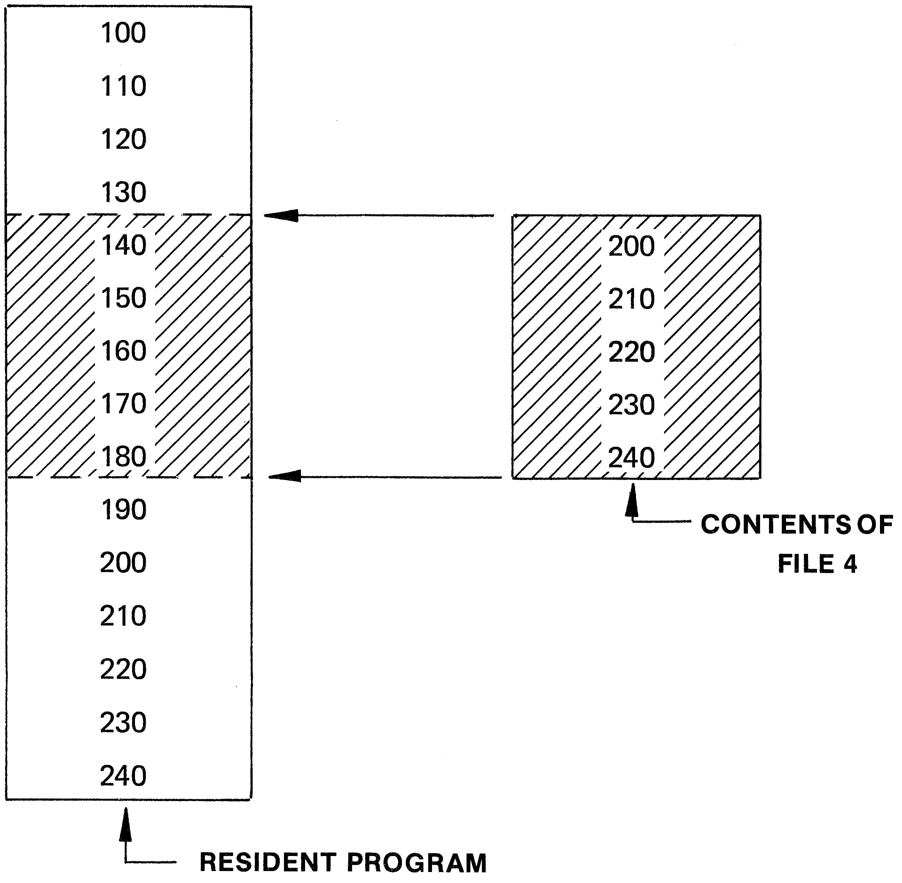
The APPEND statement also allows you to insert statements into the resident program at locations other than the end. To illustrate, assume that you have a situation similar to the previous one: a resident program and some statements in file 4 you want to insert into the program starting at statement 140. This is depicted below;



The process is essentially the same as before:

```
FIND 4
APPEND 140
```

The resident program now looks like the following:



Notice that statement 140 is now the same as the first statement in file 4, and the old statement 140 is lost. Notice also that statements 150—200 have been renumbered and retained.

The BAPPEN Routine

```
[Line number] CALL "BAPPEN",   Line number [,increment]
```

NOTE

A 4051 Graphic System must be equipped with a Binary Loader ROM Pack before using this routine.

The BAPPEN (Binary APPEND) routine inputs BASIC statements stored in binary code on tape and attaches those statements to the program currently in memory.

The BAPPEN routine follows the same procedure as the APPEND statement. First, the read/write head of the tape drive 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 tape. 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 increment 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 200 with the first statement in file 2. The remaining statements in file 2 are then appended with line numbers incremented by 10.

The LINK Routine

```
[Line number] CALL "LINK",  line number
```

NOTE

A 4051 Graphic System must be equipped with a Binary Loader ROM Pack before using this routine.

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 tape. Program execution starts at the first line number of the new program.

These features allow you to break a large BASIC program into subroutines, 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. If memory cannot hold an entire program, you can break the program into sections and use the LINK routine to load and execute the program sections.

Locating the LINK Point

To use the LINK routine, the read/write head of the tape drive 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 binary program (previously located by the FIND statement) is then loaded into memory.

If the LINK routine is performed directly from the keyboard, 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.

To use the LINK routine a BASIC program must be stored in binary code. As an example, the following program is stored in file 1 on the internal magnetic tape unit. Since this file is stored in binary code, 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.

ENVIRONMENTAL CONTROL

The earlier discussion about program control, tape control, and memory monitoring directives introduced some Graphic System BASIC statements which were shown to be quite useful for manipulating programs, but which did not help much in learning to program. In this sense, the environmental control directives are similar. In fact, these particular directives are somewhat "transparent", i.e., when you apply power to the system, the environmental conditions are set to default values, and these default values have been chosen for suitability to most applications. You can generally write programs and run them without preceding your efforts with a lot of environmental specifications, but at the same time you have the option of exercising some useful control.

The SET Statement

The most straightforward specification is that of selecting the trigonometric units of measure. When power is first applied to the system the trigonometric units are set to radians. That is, if you apply power and then type SIN(5) and press RETURN the resulting number on the display is the sine of 5 radians, not 5 degrees. If you want the sine of 5 degrees, you can cause the Graphic System to work with degrees instead of radians through the SET statement; just type SET DEGREES. The SET statement has the following general form:

```
[Line number] SET environmental condition
```

The environmental condition can be selected from four groups of conditions. One group, as already suggested, allows the selection of the trigonometric units. You can select DEGREES, RADIANS, or GRADS* . The default selection is RADIANS.

A second group enables a special operation that traces the flow of program execution. This is useful when you are "debugging" a program and want to examine the flow of the program. This second group has two conditions: TRACE and NORMAL. When you say SET TRACE and then run a program, the display contains a list of line numbers corresponding to the sequence in which the statements of the program are executed. This lets you do things such as spot infinite loops, determine if conditional transfers are working the way you envisioned them, etc. The TRACE condition is disabled when you enter SET NORMAL. Of the two conditions in this second group, the default condition is, as you would expect, NORMAL.

***Grad is defined as one one-hundredth of a right angle (90°).**

The third group of conditions associated with the SET statement involves the user definable keys. SET NOKEY causes the user definable keys to be ignored while a program is running; SET KEY causes the opposite situation. By default, the SET NOKEY condition is established when you turn on the power. The use of the user definable keys is discussed in the "Subroutines" section of this manual; the KEY/NOKEY specifications are mentioned here only because they are part of the SET statement.

The fourth group (CASE/NOCASE) is discussed in the CHARACTER STRINGS section of the manual. Briefly, SET CASE causes all characters of the alphabet, both upper and lower case, to be regarded as upper case. That is, "A" is equivalent to "a". SET NOCASE causes the reverse; i.e., "A" is not equivalent to "a". The default setting is SET CASE so that "A" = "a".

To reiterate the components of SET, the statement can be used to specify environmental conditions which occur in four groups:

Statement Form			Default Setting
[Line number]	SET	DEGREES RADIANS GRADS	RADIANS
[Line number]	SET	TRACE NORMAL	NORMAL
[Line number]	SET	KEY NOKEY	NOKEY
[Line number]	SET	CASE NOCASE	CASE

The FUZZ Statement

An additional environmental control statement allows you to establish the degree of precision the Graphic System will use when making comparisons between numbers. The BASIC interpreter compares numbers, you will recall, in IF . . .THEN . . . statements, and also makes comparisons when determining if the termination point in a FOR . . .NEXT . . . loop has been reached. Like the other environmental control statements that have already been mentioned, a default condition exists so that most of the time you need not be concerned with specifying this closeness, or comparison, value. However, the facility exists for making this specification; the statement for doing so is the FUZZ statement:

```
[Line number] FUZZ numeric expression [, numeric expression]
```

As you can see, FUZZ always has one parameter associated with it, and may have two. The first parameter specifies the number of digits that the BASIC interpreter considers as being significant when making comparisons that do not involve zero. That is, if you enter a statement like:

```
100 FUZZ 5
```

the machine only "looks at" the first 5 digits in making a comparison. If you go on to make statements like:

```
110 A=123451111
120 B=123459999
130 IF A=B THEN 200
```

statement 130 transfers control to statement 200. This happens because FUZZ is set to 5, the first 5 digits of A and B are equal, and the condition is therefore true in statement 130.

If you want to specify a closeness factor for comparisons involving zero, you enter the appropriate value via the second parameter. That is, if you want to make a comparison like:

```
210 IF Y=0 THEN 150
```

and if "almost zero" is suitable, then you can set the second parameter of FUZZ to 10^{-10} (for example). You still have to specify something for the first parameter, however, so you might write something like:

```
200 FUZZ 10,1E-10
```

With the second parameter set to 1E-10 (10^{-10}), anything that is less than 10^{-10} is considered to be zero.

There are, of course, default values for FUZZ.

The default setting is:

FUZZ 12, 1E-64

This means that, in comparisons not involving zero, only the first 12 digits are "looked at" to determine if two numbers are equal. This also means that any number less than 10^{-64} is considered to be equal to zero in comparisons like

```
150 IF A=0 THEN 200
```

These default values are adequate for most programming applications.

The INIT Statement

After reading through this discussion about FUZZ and the earlier one about SET, it might have occurred to you that you could have set the various environmental specifications to any of a number of combinations, and it is easy to lose track of them. How, you might ask, can you get everything reset to a known state? You could, of course, turn the power off and then back on: that would re-establish all the default conditions. That would also wipe out your program, however. Fortunately, there is a statement that initializes the environmental parameters to a known state; it is the INIT (for initialize) statement, formed as follows:

```
[Line number] INIT
```

When you do an INIT, either from the keyboard or under program control, here is what happens.

FUZZ is set as follows:

FUZZ 12, 1E-64

The machine examines the first 12 digits of numbers undergoing comparisons, and anything less than 10^{-64} is considered to be zero in comparisons with zero.

The DEGREE/RADIAN/GRAD selection is equivalent to the following:

SET RADIAN

All trigonometric functions will be working in radians.

The TRACE/NORMAL selection becomes

SET NORMAL

The KEY/NOKEY option becomes

SET NOKEY

The CASE/NOCASE selection becomes

SET CASE

A RESTORE statement is executed.

The assigned values of variables appearing in memory become deleted. That is, if you were to enter statements like:

```
100 LET A=10
110 INIT
120 PRINT A
130 END
```

and then type RUN, you would receive an error message informing you that you were dealing with an undefined variable.

INIT also initializes some additional parameters in areas that have not been discussed yet (particularly in Graphics); these additional parameters of INIT will be brought to your attention in the appropriate discussions.

SUMMARY

Directives are programmable statements in the BASIC interpreter repertoire which allow you to exercise control over operation of the Graphic System. RUN is probably the most often used directive—it causes the BASIC interpreter to execute the program currently in memory. LIST is also often used to examine the program in memory.

You can delete variables, specified statements, or the entire program with DELETE, and renumber it with RENUMBER. RENUMBER also lets you renumber a specified portion of a program, and it automatically updates all statement number references like those occuring in GOTO and IF . . . THEN . . . statements.

SPACE informs you of the upper bound of the number of bytes required to store the resident program. MEMORY tells you how much free memory remains. These directives are especially useful in conjunction with storing programs on tape.

You can find the beginning of a tape file with FIND. Files are created with MARK. To save a program on tape, first locate the beginning of the file with FIND, then store it with SAVE (for ASCII files) or BSAVE (for binary files).

ASCII programs are retrieved from the tape with OLD. The process parallels that of storing programs: use FIND to position the tape at the beginning of the file, then use OLD to recover the program. An OLD statement automatically executes a DELETE ALL before transferring a program from tape to memory. Binary files are retrieved from tape with the BOLD routine.

To examine the contents of the tape, execute a TLIST statement. This results in an itemization of the existing files on the tape; the information appears on the screen.

You can build a program using APPEND. This statement allows you to add-on program statements from an ASCII file at any point in the resident program. The incoming program steps must APPEND to an existing statement number, thereby "overwriting" it. Binary programs can be added to the resident program with the BAPPEN routine.

Binary programs can be loaded into memory without changing any variables or values by using the LINK routine.

The operating environment of the Graphic System can be changed from the default settings with SET and FUZZ. SET lets you specify degrees, radians, or grads. You can also specify TRACE or NORMAL. TRACE lets you examine the flow of program execution by providing a list of line numbers that are executed. SET also lets you specify KEY or NOKEY to enable the user definable keys, and the CASE/NOCASE option. FUZZ determines the "closeness value" to be used in making comparisons.

You can restore the default conditions with INIT.

EXAMPLE PROGRAM

TITLE: Directory Utility Routine

DESCRIPTION: This program is a simple "utility" routine which provides a directory of the programs stored on tape. The purpose of such a routine is to simplify the task of keeping records of what program is in what file. You store the directory utility routine in file No. 1. When you press the AUTOLOAD key, the BASIC interpreter executes the program contained in file 1, thereby providing the directory. From this, it is convenient to go directly to the desired program file.

PROGRAM LISTING:

```
100 REMARK ** DIRECTORY UTILITY ROUTINE **
110 PAGE
120 PRINT "FILE";"   CONTENTS"
130 PRINT "----";"   --------"
140 REMARK ** INSERT DIRECTORY ENTRIES BELOW THIS LINE **
150 PRINT 2;"      Name of program in file 2"
160 PRINT 3;"      Name of program in file 3"
170 REMARK **NEW ENTRIES GO IMMEDIATELY ABOVE THIS LINE**
180 PRINT
190 PRINT
200 PRINT "DO YOU WANT TO RUN A PROGRAM? (YES=1, NO=2) ";
210 INPUT A
220 GO TO A OF 230,280
230 PRINT "WHICH FILE NUMBER ";
240 INPUT A
250 PAGE
260 FIND A
270 OLD
280 PRINT "DO YOU WANT TO LIST THE DIRECTORY? (YES=1, NO=2) ";
290 INPUT A
300 GO TO A OF 310,330
310 PAGE
320 LIST
330 END
```

METHODOLOGY: The keyword PAGE is used to clear the screen under program control. Also, the program utilizes the directives FIND and OLD to retrieve program files under program control. The use of LIST in line 320 causes the program to list itself.

Actually, AUTOLOAD executes the first program file on a tape, which may or may not be the first file.

OPERATING PROCEDURE: With the directory utility routine stored in file 1 of the tape, press AUTOLOAD and the Graphic System executes file 1 and displays the directory on the screen. New directory entries (reflecting the addition of new program files on the tape) should use line numbers falling between the number of the line containing the last directory entry and the number of the line containing the REMARK which follows the directory entries. For example in the preceding program listing, a new directory entry could go at line 165. You would then type RENUMBER to resequence the line numbers, and a FIND 1, SAVE sequence to store the updated directory back on the tape.

When you press AUTOLOAD and display the directory, the routine asks "DO YOU WANT TO RUN A PROGRAM" (YES = 1, NO = 2)". If you do want to run a program, enter a 1, press RETURN, and the next prompting is "WHICH FILE NUMBER?" Enter the appropriate file number, press RETURN, and the routine finds the file, loads it into memory, and executes the program. If you didn't want to run a program, this is indicated with an entry of 2 (for "NO"); the program then asks "DO YOU WANT TO LIST THE DIRECTORY? (YES = 1, NO =2)". To list the directory routine (for example, to update the directory entries), enter 1, press RETURN, and a listing of the routine will appear. If you enter a 2 (for "NO"), the routine simply terminates.

OUTPUT SAMPLE:

```
FILE   CONTENTS
----   --------
 2     Name of program in file 2
 3     Name of program in file 3

DO YOU WANT TO RUN A PROGRAM? (YES=1, NO=2) 1
WHICH FILE NUMBER 3
```

```
FILE   CONTENTS
----   --------
 2     Name of program in file 2
 3     Name of program in file 3

DO YOU WANT TO RUN A PROGRAM? (YES=1, NO=2) 2
DO YOU WANT TO LIST THE DIRECTORY? (YES=1, NO=2) 1
```

Section 4

ARRAYS

INTRODUCTION

An array is a collection or series of data items arranged in some defined pattern. Tax tables and bus schedules might be thought of as being arrays. As far as the BASIC interpreter is concerned, an array consists of a number of locations in memory, each location capable of being identified by a unique array variable. An array "looks" something like the following:

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

The name that has been arbitrarily assigned to this particular array is "A". "A" contains, in this case, 10 elements, each possessing its own two-part identifier consisting of the name of the entire array, which is A, and a subscript enclosed within parentheses. Each element is referenced by specifying its relative position within the array. Thus, it is possible to refer to A(1), A(2), and so on. Incidentally, A(1) reads "A sub one", A(10) reads "A sub ten", etc. The ten element array A above is a "one dimensional" array — that is, its elements are stacked horizontally into one row, but there is no such stacking of elements into vertical columns.

A two-dimensional array (or "matrix") is composed of rows and columns, like the array "B" below:

	Col. 1	**Col. 2**	**Col. 3**	**Col. 4**	**Col. 5**	**Col. 6**
Row 1	B(1,1)	B(1,2)	B(1,3)			
Row 2	B(2,1)	B(2,2)				
Row 3	B(3,1)					B(3,6)
Row 4					B(4,5)	B(4,6)
Row 5				B(5,4)	B(5,5)	B(5,6)

B

As you can see, each element of a two-dimensional array can be identified by a two-part identifier as is the case with one-dimensional arrays. The only difference is that the subscript is composed of two numbers instead of one, and the numbers refer to the row and column, respectively, in which each element is contained. It is possible, therefore, to refer to each unique element of a two-dimensional array with a subscript as in B(3,1) or B(5,5) above, or more generally,

array (r,c)

where "array" represents a variable, and the subscripts r,c indicate row and column, thereby specifying the relative position of each element.

At this point, you may be wondering what programming needs are satisfied by arrays. The answer lies in the fact that programming tasks frequently involve large amounts of data, and each item of data must be stored in the memory under a variable name. Recall that the symbols avaliable for use as variables are restricted to the letters A-Z and A0 through Z9, for a total of 286 variables. This arrangement by itself doesn't permit a great deal of data to be stored.

An array, however, can contain a considerable amount of data and only requires a single name. Also, arrays permit convenient ways of referencing each item of data they contain, and BASIC provides the means to perform operations on entire arrays as though they were numeric variables. (For example, you can multiply one array by another.)

An array can be subscripted either explicitly (using constants), as in:

```
B(3)
X1(4,9)
```

or implicitly (using variables), as in:

```
C(X)
Y(R,C)
R3((N+2),(M+N))
```

Implicit subscripts are, of course, restricted to numeric expressions which are evaluated and rounded to the nearest integer. And, as you can see, the subscripts associated with a two-dimensional array are separated with commas.

ALLOCATING MEMORY

The DIM Statement

Because arrays can be of such widely varying sizes and therefore require varying amounts of memory, it is necessary to inform the Graphic System that you are going to be using arrays, and to specify the size of those arrays. This is accomplished through the use of a DIM statement. DIM is a shortened version of the word dimension. The syntax form is as follows:

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

Any valid numeric variable (A-Z and A0-Z9) may be used as the array variable, except that the same variable name can not be used as both a "regular" variable and an array variable. The numeric expressions are rounded to the nearest integer, and represent the number of rows and columns, respectively, present in a two-dimensional array. If you use a variable as a subscript, as is often the case, the variable must have an assigned value. That is, if you write a statement like

```
500 DIM A(M,N)
```

both M and N must have previously been assigned a value. You can re-dimension an array, but the total number of elements in the re-dimensioned array must be less than or equal to the number of elements originally specified.

If you want to allocate space in memory for a one dimensional array "A" containing 50 locations, a statement to use is:

```
100 DIM A(50)
```

A statement that establishes an array "B" containing 5 rows and 15 columns is:

```
150 DIM B(5,15)
```

More flexible allocation methods can be obtained through the combined use of INPUT or READ/DATA and DIM. For example, you might know that a particular program is going to require a one dimensional array to contain some data, but the size is uncertain or is apt to change from one run to another. The program can be written to include a segment like:

hard copy goes here

```
500 INPUT N
505 DELETE A
510 DIM A(N)
```

This way, you allocate space at the time you run the program, and the size of the array can be adjusted to suit the situation. The DELETE statement is included to permit "upward re-dimensioning." That is, you might run the program once and dimension A to contain 100 elements, and then run it again and dimension A to contain 250 elements. Bearing in mind that you can only redimension downward, the solution is to delete the variable entirely from memory, and then re-allocate a new space. Two-dimensional arrays can, of course, be handled similarly:

```
300 PRINT "ENTER NUMBER OF ROWS ";
310 INPUT R
320 PRINT "ENTER NUMBER OF COLUMNS ";
330 INPUT C
340 DIM A(R,C)
```

If the program is to include READ and DATA statements, then the allocation process is essentially the same, as in:

```
500 READ R,C

510 DIM A(R,C)

     .

     .

     .

     .

900 DATA 10,20
```

In this case, R receives the value of 10, and C receives the value of 20, thereby dimensioning "A" to be 10 rows and 20 columns.

NOTE

Arrays are limited to two dimensions.

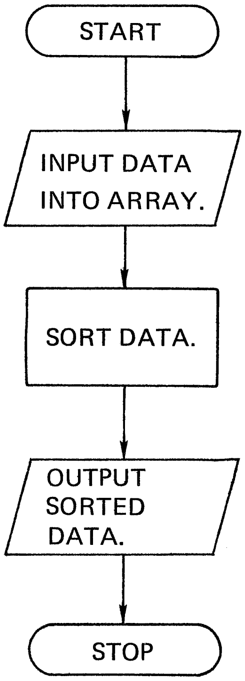
One more thing about allocating memory space: you can dimension several arrays in one statement. For instance:

```
340 DIM A(R,C),B(X),C(15)
```

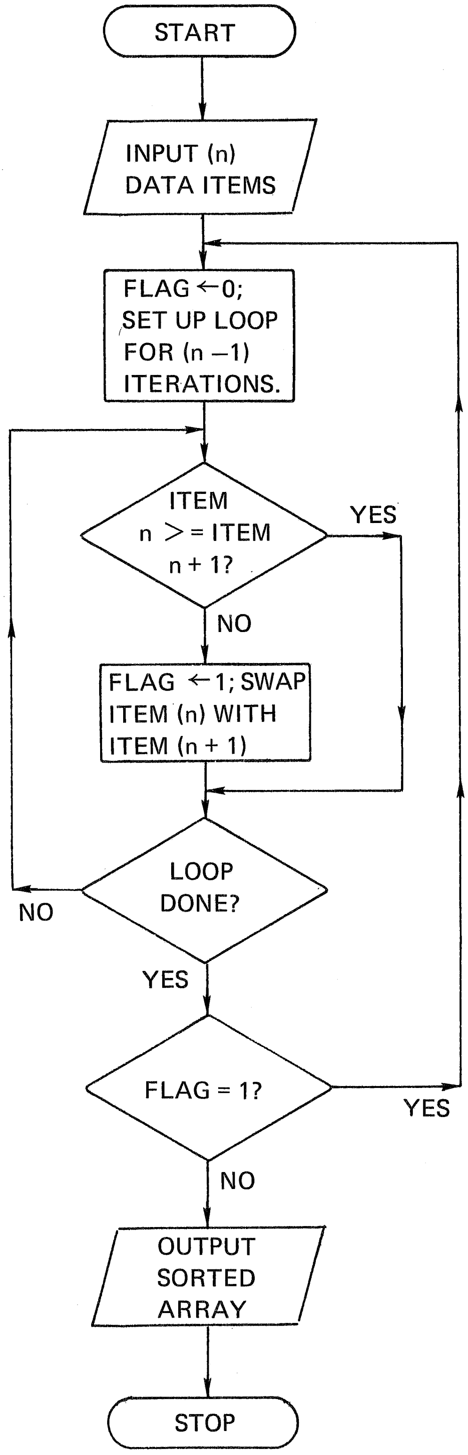
SUBSCRIPTING

One Dimensional Arrays

Let's suppose that you have n numbers to sort into descending order. Obviously, you are going to have to get these numbers into memory, and an array is going to be the best place to store them. Once the n numbers are stored, there remains the question of how to sort them. A good place to begin such an exercise is with a flowchart that organizes the method to be used. An initial attempt at generating this flowchart might yield something like the one shown below. Closer examination of this initial flowchart, however, suggests that another level of detail must be added; the program cannot be adequately envisioned from what exists so far. The input and output program segments can be written using a FOR/NEXT loop and an INPUT statement for the "input" step; another FOR/NEXT loop can appear with a PRINT statement to take care of the "output"step. However, more detail is required to accomplish the process indicated by the "sort data"step, because the method still remains to be established.



The second flowcharting effort (below) gives sufficient detail to permit writing the program. The "input" and "output" steps remain the same, but the "sort data" process has been expanded considerably.



Using a loop, the data is brought into a one-dimensional array with n elements.

The flag is used to indicate whether or not any out-of-sequence data has been encountered in the array; the loop is used for subscripting the array.

Determine if the adjacent pair (items n and n + 1) are in correct order. If they are, then bypass the procedure that swaps them around.

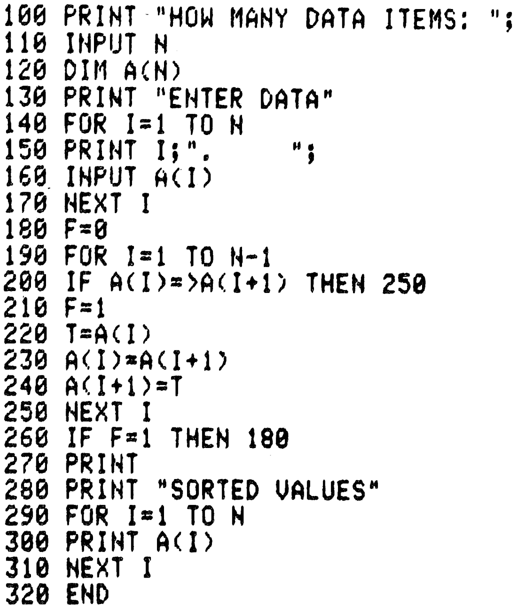
An adjacent pair is found to be in reverse order. Interchange them, and set the flag to indicate that a swap has been made and that more may be necessary.

If the loop (used for subscripting the array variable) has terminated, then proceed; otherwise, return to the beginning of the loop.

The loop is done; examine the flag to see if any swaps have occurred. If some swaps were performed, go back through the data again because more swaps may be required.

Print out the original data and the sorted data.

The program, written from the flowchart, is as follows:

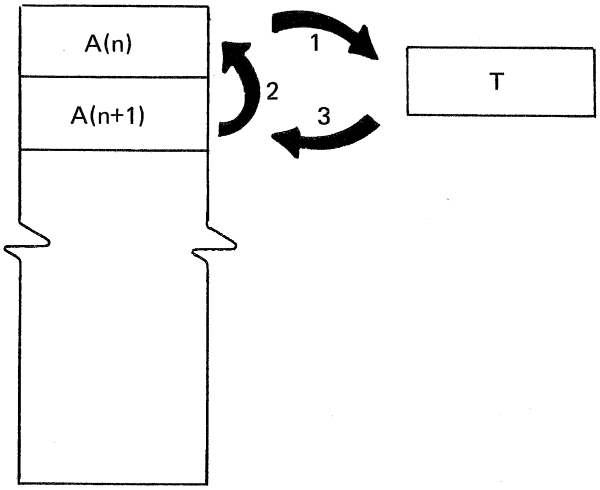


```
100 PRINT "HOW MANY DATA ITEMS: ";
110 INPUT N
120 DIM A(N)
130 PRINT "ENTER DATA"
140 FOR I=1 TO N
150 PRINT I;".      ";
160 INPUT A(I)
170 NEXT I
180 F=0
190 FOR I=1 TO N-1
200 IF A(I)=>A(I+1) THEN 250
210 F=1
220 T=A(I)
230 A(I)=A(I+1)
240 A(I+1)=T
250 NEXT I
260 IF F=1 THEN 180
270 PRINT
280 PRINT "SORTED VALUES"
290 FOR I=1 TO N
300 PRINT A(I)
310 NEXT I
320 END
```

The array which contains the n data items is "A"; allocation of memory for "A" is accomplished in lines 100—120. The data items which are to be sorted are read into "A" with lines 130—160. Notice that a FOR/NEXT loop is utilized, and the values that the loop variable "I" assumes are used as subscripts in statement 160. This particular FOR/NEXT loop repeats until each element of "A" is filled.

The variable "F" in statement 180 is the flag which appears in the flowchart; setting it equal to zero "clears" the flag. The actual data sorting within array "A" is accomplished in the FOR/NEXT loop statements 190—250.

The operation of the sorting routine relies on the technique of comparing the values of adjacent pairs. The first time through the loop, item (1) is compared with item (2). If item (1) is greater than or equal to item (2), then they are left in their current order and program control passes to statement 250. At statement 250, the loop variable "I" is incremented to its next value, and the next iteration of the loop is executed. If, however, item (1) is less than item (2), item (1) is placed in a temporary location "T", item (2) is placed in item (1)'s location, and the contents of the temporary location "T" are placed in item (2)'s location. Pictorially, the process looks like this:



Also, if item (1) is less than item (2), then the flag is set; i.e., "F" is assigned the value of one. This flag is only set if a swap is to be performed. When the flag is set, it indicates that an out-of-sequence pair has been discovered, and that the entire array "A" has to be examined again to see if anything remains out of sequence. The flag is examined at statement 260; if it is set, control transfers to line 180. Once control reaches line 180, the flag is cleared, and the whole sorting procedure is repeated. Eventually, all the data in array "A" is correctly sequenced, the flag is cleared, and program control reaches statement 270.

Once program control reaches line 270, the output of the program is produced. A FOR/NEXT loop is again utilized for subscripting, and the output is generated by the PRINT statement within the body of the loop. When you run the program, it asks you for input and then produces output as indicated below.

```
HOW MANY DATA ITEMS: 10
ENTER DATA
 1.      23
 2.      45
 3.      6
 4.      7
 5.      56
 6.      23
 7.      45
 8.      67
 9.      78
10.       8
```

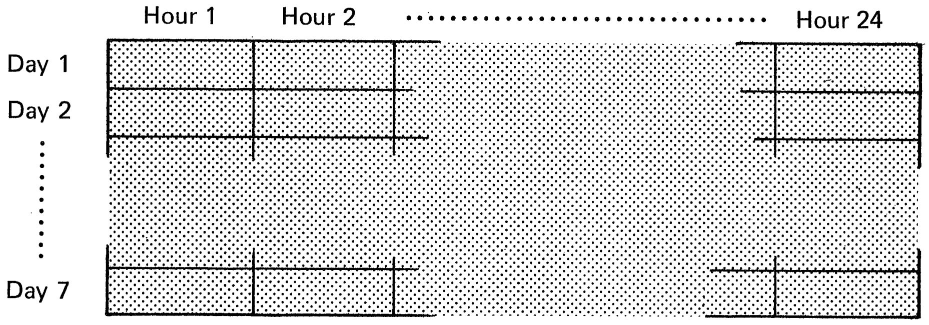
```
SORTED VALUES
 78
 67
 56
 45
 45
 23
 23
 8
 7
 6
```

The important thing to be gained from this exercise is the exposure to some methods of subscripting arrays. The sorting technique is not a particularly efficient one, and the program itself can be written with fewer lines. The main thing is, again, the manipulations that are made possible by subscripting.

Two Dimensional Arrays

The earlier discussion about one dimensional arrays centered around a sorting program. That program employed a FOR/NEXT loop to input data into an array; data was assigned to the array location specified by the loop-generated subscript. A similar procedure can be used with two dimensional arrays. The difference is that two dimensional arrays require two subscripts instead of one; this suggests that two FOR/NEXT loops are required, one for each subscript.

To expand on this idea, suppose that you have been collecting hourly temperature readings for a period of one week as a part of a project. A reasonable format for this data is to arrange it into seven rows (one for each day of the week) and twenty-four columns (one for each hourly reading). This, in turn, suggests a 7 X 24 array:



To enter this data into the Graphic System, the first thing to do, of course, is to dimension (or allocate) the array:

```
100 DIM T(7,24)
```

This sets up the seven row, twenty-four column matrix. Now, in order to subscript each element of the temperature array "T", two FOR/NEXT loops need to be established. One will run from 1 through 7 (for rows), and the other will run from 1 through 24 (for columns). A nested loop technique will supply the subscripts. For clarity, the outer loop will use the variable R for rows; the inner loop will use C for columns. The following program segment, incorporating these ingredients, stores the data in the array T:

```
110     FOR R = 1 TO 7
120       FOR C = 1 TO 24
130       INPUT T(R,C)
140       NEXT C
150     NEXT R
```

When the first repetition of the outer loop takes place, R = 1 and C = 1, and statement 130 assigns data to T(1,1). Then, while R remains at 1, C goes to 2, and the next data item is assigned to T(1,2). In this manner, row 1 is filled, then row 2, etc. Finally, each data item is assigned to an element of the array, and the outer loop terminates. The data are now organized and stored, and are easily accessible for some further processing.

Incidently, you might have noticed that the subscripts in statement 130 above are arranged so that the outer loop variable is specified first, followed by the inner loop variable. The result is that the array is "filled" row-by-row. By reversing the order of these variables within the subscript, it is possible to reference an array column-by-column. This is shown below, using "O" as the outer variable, "I" as the inner variable, and a hypothetical 4 X 3 array "H".

```
500     FOR O = 1 TO 3
510       FOR I = 1 TO 4
520       READ H(I,O)
530       NEXT I
540     NEXT 0
```

This program segment causes the variables and subscripts to assume values as tabulated below, with the indicated effect on the array.

O	I	(I,O)
1	1	1,1
1	2	2,1
1	3	3,1
1	4	4,1
2	1	1,2
2	2	2,2
2	3	3,2
2	4	4,2
3	1	1,3
3	2	2,3
3	3	3,3
3	4	4,3

Contents of array

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

The main thing to be aware of here is that you can reference an array either column-by-column or row-by-row, depending on the manner in which the subscripts are arranged.

Brief Examples of Array Operations

The following program segments are included to increase your familiarity with array operations.

Procedure to sum the elements of a one-dimensional array.

Given:

(1) Array name is A.

(2) A contains N defined elements (each element contains a value).

(3) Sum is to be stored in S.

Procedure:

```
100 S=0
110 FOR I=1 TO N
120 S=S+A(I)
130 NEXT I
140 PRINT "SUM= ";S
```

Procedure to sum the elements of a two-dimensional array.

Given:

(1) Array name is A.

(2) A contains R rows and C columns.

(3) Each element of A is defined.

(4) Sum is to be stored in S.

Procedure:

```
100 S=0
110 FOR I=1 TO R
120 FOR J=1 TO C
130 S=S+A(I,J)
140 NEXT J
150 NEXT I
160 PRINT "SUM= ";S
```

Procedure to sum the elements of a two-dimensional array by row.

Given:

(1) Array name is A.

(2) A contains R rows and C columns.

(3) Each element of A is defined.

(4) A one-dimensional array S is allocated with R elements (one for each row of A). Each element of S has been initialized to zero. The array S contains the sums of the rows in array A.

Procedure:

```
100 FOR I=1 TO R
110 FOR J=1 TO C
120 S(I)=S(I)+A(I,J)
130 NEXT J
140 NEXT I
150 FOR I=1 TO R
160 PRINT "SUM OF ROW ";I;" IS ";S(I)
170 NEXT I
```

Procedure to copy the contents of a two-dimensional array into a one-dimensional array.

Given:

(1) Array name is A.

(2) A contains R rows and C columns.

(3) Each element of A is defined.

(4) A one-dimensional array B is allocated with (R * C) elements.

Procedure:

```
100 K=0
110 FOR I=1 TO R
120 FOR J=1 TO C
130 K=K+1
140 B(K)=A(I,J)
150 NEXT J
160 NEXT I
```

Comment: The variable K is used as a counter for subscripting array B.

Procedure to find the mean, standard deviation, sum of the squares, and sum of the observations of the contents of a one-dimensional array.

Given:

(1) Array name is A.

(2) A contains N elements.

(3) Each element of A is defined.

(4) Sum of squares (ΣX^2) is stored in S.

(5) Sum of observations (ΣX) is stored in S1.

Procedure:

```
100 S=0
110 S1=0
120 FOR I=1 TO N
130 S=S+A(I)↑2
140 S1=S1+A(I)
150 NEXT I
160 S2=SQR((N*S-S1↑2)/(N*(N-1)))
170 PRINT "STANDARD DEVIATION= ";S2
180 PRINT "SUM OF SQUARES= ";S
190 PRINT "SUM OF OBSERVATIONS= ";S1
200 PRINT "MEAN= ";S1/N
```

OPERATIONS

Array Operations

The procedures which were discussed earlier concerning the input of data into an array and the output of data from an array were primarily intended to be exercises for the development of subscripting concepts. Fortunately, BASIC provides much easier methods of accomplishing array input/output; also, a number of built-in functions that pertain to arrays are included in the language. The following paragraphs describe the kinds of array operations that are possible with the Graphic System.

Inputting Arrays

It is possible to input data to an array from the keyboard without having to set up FOR/NEXT loops to do so. The statement that does this is the conventional INPUT statement:

```
[Line number] INPUT array variable [, array variable] . . .
```

The only difference between this statement and the INPUT that has appeared in previous discussions is the fact that an array variable is involved rather than a numeric variable. In order to use the INPUT statement in this manner, it is necessary to first inform the BASIC interpreter that you are going to be using an array; this, of course, is accomplished through a DIM statement. Also, it is important to know that the INPUT statement assigns data to the elements of a matrix in row-major order (i.e., the first row is filled first, then the second row, etc.).

When the BASIC interpreter encounters an INPUT statement for an array, it generates a question mark as before, indicating that it is waiting for data entry from the keyboard. A question mark is generated for each element of an array.

As an example, the following statements permit you to input data into a 5 X 10 array:

```
100 DIM A(5,10)
110 INPUT A
```

The execution of statement 110 (above) places a question mark on the screen. You then enter the appropriate data values, separating each value with a comma or a RETURN. (If you use commas, use a RETURN at the end of the last data value.) Question marks continue to appear until each element of the array has been assigned a value.

Reading Arrays

The READ statement, like INPUT, allows you to fill an array without having to use FOR/NEXT loops. Again, the specified array is filled in row major order. The main difference, of course, is that the data is obtained from a DATA statement instead of from the keyboard. The syntax form is:

```
[Line number] READ array variable [, array variable] . . .
```

An example:

```
100 DIM A(3,4)
110 READ A
500 DATA 1,2,3,4,5,6,7,8,9,10,11,12
```

This example reads data into array A as follows:

	Col. 1	Col. 2	Col. 3	Col. 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

Printing Arrays

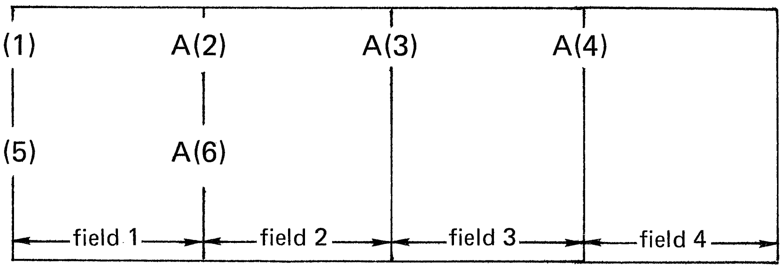
As you would expect, this statement prints out the contents of the specified array in row major order. The syntax form for this statement is:

```
[Line number] PRINT array variable [, array variable] . . .
```

The format of the output is dependent on the screen's four print fields. That is, an array such as

A(1) A(2) A(3) A(4) A(5) A(6)

is printed out on the screen in a spacing arrangement similar to that shown here:



A PRINT statement pertaining to arrays and ending with a semicolon behaves like a conventional PRINT ending with a semicolon: each element of the array is printed exactly as it is represented internally, including the leading blank.

Array Assignments

Matrix assignment statements can assume a number of different forms, according to the type of operation taking place. All the forms have included within them the assignment operator ("equals" sign), hence the term "Matrix assignments". The various aspects of matrix assignments are discussed in the following paragraphs.

An *array assignment* statement can be used to make a copy of an existing array, using the form:

```
[Line number] [LET] array variable = array variable
```

For example, given that B is a previously defined array, the statement

```
100 LET A=B
```

copies B into A. Remember that the "=" sign is the assignment operator, and the statement can be read "matrix A receives matrix B". Matrix B remains unchanged. Matrix A must have been previously given dimensions equal to those of B.

You can also initialize all the elements of an array to one value, using the form

```
[Line number] [LET] array variable = numeric expression
```

This means that you can write statements like

```
100 LET A=9
110 LET B=4*X↑2
```

Array Arithmetic

Elementary array arithmetic can be accomplished using the form:

$$\text{[Line number] [LET] array variable = array variable} \begin{Bmatrix} + \\ - \\ * \\ / \end{Bmatrix} \begin{Bmatrix} \text{array variable} \\ \text{numeric expression} \end{Bmatrix}$$

Adding Arrays

Array addition performs element-by-element summations. For example, the statement

```
100 LET A=B+C
```

sums each element of B with each element of C and places the result in A. You can also write statements like the following:

```
100 LET A=A+B
110 LET A=A+4
120 LET A=B+(3*X↑2)
```

where A and B are identically dimensioned arrays.

Subtracting Arrays

The *array subtraction* statement takes a form similar to that used with array addition, the only difference being the operator that is used. To illustrate,

```
100 LET A=B-C
```

subtracts each element of C from each element of B, and assigns the result to A. A, B, and C must be identically dimensioned. As was the case with array addition, you can also say:

```
100 LET A=A-B
```

This causes each element of B to be subtracted from each element of A; the resultant difference will reside in A. Also allowed are statements like:

```
100 LET A=B-4
```

which subtracts 4 from each element in B and assigns the result to A and, similarly,

```
100 LET A=A-SQR(X)
```

which subtracts the square root of the numeric variable X from each element of A.

Multiplying Arrays

Array multiplication is similar to array addition and subtraction. That is, you can say

```
100 LET A=B*C
```

and you can also write

```
100 LET A=A*B
```

Dividing Arrays

Array division is accomplished with a format similar to array addition, subtraction, and multiplication, and is also performed element-by-element.

Summing an Array

It is convenient to sum the elements of an array with the following function:

SUM array variable

The SUM function returns a value which is equal to the sum of all the elements of the specified array. The function can be used in numeric expressions just like SIN, COS, etc.

MATRIX FUNCTIONS

NOTE

In order to perform the matrix functions a 4051 Graphic System must be equipped with a Matrix Functions ROM Pack.

The DET Function

The DET function returns the value of the determinant:

```
   [ Line number ] array variable = INV array variable
[  [ Line number ]   numeric variable =  ] DET
```

NOTE

As indicated in the syntax form above, the INV 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.

The determinant of a matrix is a function of the elements in the matrix. The value of the determinant is a numeric constant, and can be computed for any square matrix.

NOTE

The following discussion is provided to illustrate what the determinant of a matrix is, by showing one method of computing its value by hand. This is not an explanation of the algorithm the DET function uses to compute the value of the determinant.

The value of the determinant is a numeric constant, and can be computed by hand from the elements of the array. For example:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad , \quad \text{DET} = -2$$

For a 2x2 matrix as in this example, the value of the determinant is the product of the diagonal elements in the upper-left and lower-right corners, minus the product of the elements on the other diagonal:

$$\text{DET} = 1*4 - 3*2 = -2$$

Computing the determinant of a 3x3 matrix by hand is a little harder.

For example:

$$A = \begin{bmatrix} -3 & 1 & 2 \\ 6 & 0 & 3 \\ 2 & -1 & 0 \end{bmatrix}, \quad \text{DET} = -15$$

One way to begin the calculation is to "expand" along the first row as follows: multiply each element along the first row by the determinant of the matrix found by covering up the row and column containing that particular element. For instance, multiply the element A(1,1) by the determinant of the matrix found by covering up the first row and first column of A:

$$\begin{bmatrix} -3 & 1 & 2 \\ 6 & 0 & 3 \\ 2 & -1 & 0 \end{bmatrix}, \quad -3*[0*0-(-1)*3] = -9$$

Notice that the determinant of the 2x2 matrix $\begin{bmatrix} 0 & 3 \\ -1 & 0 \end{bmatrix}$ is obtained by the method described in the first example, that is $0*0-(-1)*3=3$.

Next, multiply the element A(1,2) by the determinant of the matrix found by covering up the first row and second column of A:

$$\begin{bmatrix} -3 & 1 & 2 \\ 6 & 0 & 3 \\ 2 & -1 & 0 \end{bmatrix}, \quad 1*[6\cdot 0-2*3] = -6$$

Finally, multiply A(1,3) by the determinant of the matrix found by covering up the first row and third column of A:

$$\begin{bmatrix} -3 & 1 & 2 \\ 6 & 0 & 3 \\ 2 & -1 & 0 \end{bmatrix}, \quad 2*[6*-1-2*0] = -12$$

Three numbers are obtained in this way, −9, −6, and −12. The value of the determinant of A is the first number, minus the second number, plus the third number:

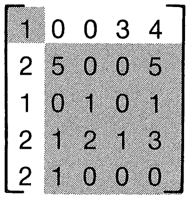
$$\text{DET} = -9 + 6 - 12 = -15$$

The same method can be used to find the determinant when A is a larger matrix. Each element along the first row of A is multipled by the determinant of the matrix found by covering up the row and column containing that element. The value of the determinant of A is equal to the first number, minus the second number, plus the third number, minus the fourth number, and so on.

The computations are tedious for matrices having more than three rows and three columns. For example, take A to be the 5x5 matrix shown below:

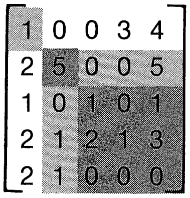
$$A = \begin{bmatrix} 1 & 0 & 0 & 3 & 4 \\ 2 & 5 & 0 & 0 & 5 \\ 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 2 & 1 & 3 \\ 2 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad \text{DET} = -43$$

The first step in computing the determinant is to multiply A(1,1) by the determinant of the matrix found by covering up the first row and first column of A:



$$\begin{bmatrix} 1 & 0 & 0 & 3 & 4 \\ 2 & 5 & 0 & 0 & 5 \\ 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 2 & 1 & 3 \\ 2 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Now to compute the value of the determinant of the 4x4 matrix shaded above, begin by multiplying the first element in its first row by the determinant of a 3x3 matrix:



$$\begin{bmatrix} 1 & 0 & 0 & 3 & 4 \\ 2 & 5 & 0 & 0 & 5 \\ 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 2 & 1 & 3 \\ 2 & 1 & 0 & 0 & 0 \end{bmatrix}$$

This process continues until all of the required "subdeterminants" have been reduced to a numeric value. Notice that **each** element in the first row of matrix A must be multiplied by a 4x4 determinant, whose value must be reduced to a numeric constant by performing the necessary expansions on 3x3 matrices, and so on.

It is easy to see the advantage of using the DET function to compute the determinant of such a matrix; the DET function calculates the value of the determinant in a fraction of the time it takes to complete the calculation by hand.

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:

$$A=\begin{bmatrix} 1 & 1 & 0 \\ 1 & 2 & -3 \end{bmatrix}, \quad B = \mathrm{INV}(A) = \begin{bmatrix} 2 & -1 & 3 \\ -1 & 1 & -3 \end{bmatrix}, \quad \mathrm{DET} = 1$$

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 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.) In either case, the INV function must be performed on the matrix before performing the DET function.

The following program illustrates how the DET function can be used to find the determinant of the square part of a matrix:

```
100 DEL A,B
110 DIM A(2,3),B(2,3)
120 READ A
130 DATA 1,1,0, 1,2,—3
140 B = INV(A)
150 PRINT "A =";A;
160 PRINT "DET=";DET
```

This program computes the determinant of the square part (the shaded portion) of the following matrix:

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 2 & -3 \end{bmatrix}$$

The INV function is performed on matrix A in line 140, and the DET function is used in line 160. When line 160 is executed, the value of the determinant appears on the display:

DET = 1

The important thing to keep in mind is that the INV function is always performed first, then the DET function.

The DET function always returns the determinant of the square part of the matrix most recently supplied to the INV function. This means that as soon as the INV function is performed on a new matrix, the result of the DET function changes.

To store the value of the determinant for later use, the result of the DET function may be assigned to a numeric variable in a statement such as X = DET or 100 X = DET. The numeric variable is called the target variable because it serves as a "target" for the value of the determinant.

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 on 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 IDN Routine

The IDN function creates a matrix whose elements are 1's along the major diagonal, and 0's

```
[ Line Number ] CALL   "IDN"  , array variable
```

For example, the IDN function 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 function 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, if I is to be the target for the result of the IDN function, and is dimensioned in a DIM statement to be a 5x5 matrix, performing the IDN function results in the 5x5 identity matrix shown here:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

A matrix does not have to have a square shape to be used as the target for the IDN function. The IDN function 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.

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 is dimensioned to be a non-square matrix, the IDN function 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, let matrix I be dimensioned as a 6x3 array:

```
110 DIM I(6,3)
```

When the IDN function is performed, matrix I is assigned the following values:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The shaded area shown above is the square part of matrix I because it is the largest square that can be blocked off, starting in the upper-left corner. Notice that the square part is the 3x3 identity matrix, and all elements outside the square part are 0's.

When the target variable has a non-square shape, the result of the IDN function 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.

Before performing the IDN function, 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 DIM function, 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 function is performed.

Any valid numeric variable name may be used as the target for the result of the IDN function. An array can serve as the target for the IDN function without having numeric values assigned to each element. The only requirement for the target variable is that it be previously dimensioned in a DIM statement, using two subscripts to indicate the number of rows and columns in the result matrix.

An array variable name which has numeric values assigned to each element may also be used as the target variable. However, the result of the IDN function replaces all previously assigned values.

The following program illustrates how the IDN function is used to create the 4x4 identity matrix:

```
100 DELETE I
110 DIM I(4,4)
120 CALL "IDN", I
130 PRINT "I = ";I
```

Only four statements are needed to form the identity matrix and display the result. In line 110, the variable I is dimensioned to be a square matrix having four rows and four columns. Matrix I is to be used as the target for the result of the IDN function, and must be previously dimensioned in a DIM statement, or an error occurs. When the IDN function is performed in line 120, matrix I becomes the 4x4 identity matrix; 1's are assigned to the diagonal elements of I, and 0's are assigned to all other elements. When line 130 is executed, the result appears on the display as follows:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix I is now the 4x4 identity matrix.

The dimensions of matrix I may be changed in order to make an identity matrix of a different size. For example, the following program creates the 6x6 identity matrix:

```
100 DELETE I
110 DIM I(6,6)
120 CALL "IDN",I
130 PRINT "I=";I
```

Line 110 contains the only difference between this program and the program that was used to make the 4x4 identity matrix. The DIM statement in line 110 determines which identity matrix is generated by the program. Here, I is dimensioned to be a 6x6 array, so the program makes the 6x6 identity matrix shown below:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

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:

$$\text{A MPY I} = \text{I MPY A} = \text{A}$$

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:

$$\text{A} = \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix} \quad , \text{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Multiplying A by I (or I by A) result in the same matrix A:

$$\text{A MPY I} = \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix} = \text{A}$$

$$\text{I MPY A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix} = \text{A}$$

(Refer to the matrix function MPY for how to perform these caluclations.)

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 function 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 Matrix Mulitplication

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:

$$\text{A MPY B} = \text{B MPY A} = \text{I}$$

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 INV Function

The INV function returns the inverse of a matrix, and solves system of linear equations:

```
[ Line number ] array variable = INV array variable
```

The 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. A detailed explanation of what this means is given later. Meanwhile, an example is given to illustrate what the INV function does:

$$A = \begin{bmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 0 \end{bmatrix} \quad , \quad B = \text{INV}(A) = \begin{bmatrix} 2 & -3 & -7 & 8 \\ -1 & 2 & 5 & -4 \end{bmatrix}$$

Notice that when the 2x4 matrix A is supplied to the INV function, the result B = INV(A) is also a 2x4 matrix.

The Square Part

The square part of matrix B is the inverse of the square part of matrix A. The square parts of A and B are the shaded portions shown below:

$$A = \begin{bmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 0 \end{bmatrix} \quad , \quad B = \text{INV}(A) = \begin{bmatrix} 2 & -3 & -7 & 8 \\ -1 & 2 & 5 & -4 \end{bmatrix}$$

The shaded portion in each matrix is called the square part because it is the largest square that can be blocked off, starting in the upper left-hand corner of the matrix.

The shaded portion in B=INV(A) is the inverse of the shaded portion in A:

$$\begin{bmatrix} 2 & -3 \\ -1 & 2 \end{bmatrix} \quad \text{is the inverse of} \quad \begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}$$

Saying that one matrix is the inverse of another means that when the two are multiplied, the result is an identity matrix. (Refer to the IDN Function topic "Properties of Identity Matrices".)

The Extra Columns

In the previous example, both matrices A and B have two columns which lie outside the square part. The extra columns are shown below:

$$A = \begin{bmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 0 \end{bmatrix}, \quad B = \mathrm{INV}\ (A) = \begin{bmatrix} 2 & -3 & -7 & 8 \\ -1 & 2 & 5 & -4 \end{bmatrix}$$

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 $y = -4$.

The coefficients of x and y are the same for both sets of linear equations, and can be found in the square part of matrix A:

$$A = \begin{bmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

The constants which appear to the right of the equal sign in the first set of equations, appear in the first column to the right of the square part in A:

$$A = \begin{bmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

Likewise, the constants which appear in the **second** set of equations, are found in the **second** column to the right of the square part in A:

$$A = \begin{bmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

The solution for the first set of equations is found in the first column to the right of the square part in B:

$$B = \mathrm{INV}(A) = \begin{bmatrix} 2 & -3 & -7 & 8 \\ -1 & 2 & 5 & -4 \end{bmatrix}$$

The solution for the **second** set of equations is found in the **second** column to the right of the square part in B:

$$B = \mathrm{INV}(A) = \begin{bmatrix} 2 & -3 & -7 & 8 \\ -1 & 2 & 5 & -4 \end{bmatrix}$$

In summary, each extra column in B=INV(A); represents the solution to a set of equations which uses the elements of the square part in A for the coefficients of x and y. For every extra column in B, there is a corresponding extra column in A which contains the constants that appear to the right in the set of equations. Fig. 2-1 summarizes the information obtained from this example.

NOTE

For those who are accustomed to representing systems of linear equations in matrix form, it may be more convenient to express the equations and solutions as follows:

First System of Equations

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

Solution

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -7 \\ 5 \end{bmatrix}$$

Second System of Equations

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -8 \\ 4 \end{bmatrix}$$

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. For example:

$$A = \begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \quad B = \mathrm{INV}(A) = \begin{bmatrix} 3 & 2 & -6 \\ 1 & 1 & -2 \\ -1 & -1 & 3 \end{bmatrix}$$

The shaded portions shown above are the square parts of matrices A and B. Notice that the square part of A is the entire matrix A, and the square part of B is all of matrix B.

When the INV function is performed, the square part of the result is the inverse of the square part of the original matrix. This means that for this example, matrix B is the inverse of matrix A.

When the INV function is performed, extra columns in the result represent solutions to sets of linear equations. But in this example, there are no extra columns to the right of the square part of A or B, so the result does not provide solutions to sets of linear equations.

To summarize, when the INV function is performed, the square part of the result is the inverse of the square part of the original matrix, and any remaining columns represent solutions to sets of linear equations. When the matrix supplied to the INV function has a square shape, there are no extra columns, and the resulting matrix is the inverse of the original matrix.

The parameter of the INV function must be a matrix having at least as many columns as rows. For example, the INV function can be performed on the following matrices:

$$\begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix} , \quad \begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & 4 \end{bmatrix} , \quad \begin{bmatrix} 1 & 2 & 3 & 0 \\ 0 & -1 & 4 & 6 \end{bmatrix}$$

If a matrix is to be used as a parameter for the INV function, it must previously be dimensioned in a DIM statement, using two subscripts to indicate the number of rows and columns. For example, the statement DIM A(3,4) dimensions a matrix which may be used as a parameter for the INV function.

Since the matrix must have at least as many columns as rows, the second subscript used to dimension the parameter must be greater than or equal to the first subscript, or an error will occur. For example, if matrix A is dimensioned to be a 4x3 matrix in the statement DIM A(4,3) attempting to perform INV(A) results in an error.

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. The target variable 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. For example, the following statement can be used:

```
A = INV(A)
```

However, when the statement is executed, the original matrix A is replaced by the new matrix generated by the INV function. Care should be taken not to allow the original matrix A to be overwritten in this way, if A is to be used in later calculations. In general, it is good practice to use different names for the target variable and the parameter variable.

If a matrix has been overwritten by the result of the INV function, the original matrix may be recovered by performing the INV function again; however, truncation may occur during the calculations, causing the answer to be different from the original matrix. How closely the answer resembles the original matrix depends upon the "sensitivity" of the parameter matrix. A matrix is called "sensitive" if the values assigned to its elements make the matrix more susceptible to the truncation errors that normally occur when arithmetic operations are performed.

Matrices which are very sensitive to truncation error during the INV process are called "ill-conditioned" with respect to inversion. For discussion of truncation and the condition of a matrix, refer to the Supplementary Information at the end of this section.

The following program illustrates how the INV function may be used to invert a matrix and solve a set of linear equations:

```
100 DEL A,B
110 DIM A(3,4),B(3,4)
120 READ A
130 DATA 1,0,1,0,2,−1,−2,3,−2,1,1,0
140 B = INV(A)
150 PRINT "B = INV A =";B
```

This program performs the INV function for the following 3x4 matrix:

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 2 & -1 & -2 & 3 \\ -2 & 1 & 1 & 0 \end{bmatrix}$$

Line 140 computes INV(A) and assigns the result to the target variable B. When line 150 is executed, the result appears on the display:

$$B = \mathrm{INV}(A) = \begin{bmatrix} 1 & 1 & 1 & 3 \\ 2 & 3 & 4 & 9 \\ 0 & -1 & -1 & -3 \end{bmatrix}$$

The square part of the resulting matrix B is the inverse of the square part of the original matrix A:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 3 & 4 \\ 0 & -1 & -1 \end{bmatrix} \quad \text{is the inverse of} \quad \begin{bmatrix} 1 & 0 & 1 \\ 2 & -1 & -2 \\ -2 & 1 & 1 \end{bmatrix}$$

The fourth column of matrix B represents the solution for the set of linear equations shown below:

$$\begin{array}{rrrl} x & & +z & = 0 \\ 2x & -y & -2z & = 3 \\ -2x & +y & +z & = 0 \end{array}$$

The coefficients of x, y and z in the equations are the elements found in the square part of A, and the constants which appear on the right hand side of the equations are found in the last column of A.

The solution to the set of equations is represented by the last column of matrix B, as follows:

$$\begin{array}{l} x = \ \ 3 \\ y = \ \ 9 \\ z = -3 \end{array}$$

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 square part of the parameter matrix has no inverse, an INVALID FUNCTION ARGUMENT error message appears on the display. For instance, attempting to perform the INV function results in an error for any of the matrices shown below:

$$\begin{bmatrix} 2 & -2 \\ -5 & 5 \end{bmatrix}, \begin{bmatrix} 2 & -2 & 0 \\ -5 & 5 & 1 \end{bmatrix}, \begin{bmatrix} 2 & -2 & 3 & -1 \\ -5 & 5 & 0 & 9 \end{bmatrix}, \begin{bmatrix} 2 & -2 & 1 & 2 & 0 \\ -5 & 5 & 3 & 4 & 5 \end{bmatrix}$$

For each of these matrices, the square part (the shaded portion) is the 2x2 matrix C from the last example. Since C does not have an inverse, attempting to perform the INV function on any of the above matrices results in an INVALID FUNCTION ARGUMENT error message. However, if the error results while a program is executing, the BASIC interpreter behaves as if a SIZE ERROR has occurred, so that ON SIZE THEN... statement may be used to handle the error without terminating program execution.

When an ON SIZE THEN... statement is executed in a BASIC program, subsequent SIZE ERRORs do not halt program execution; instead, control is diverted to the line number specified in the ON SIZE THEN... statement. (Refer to Section 7 for an explanation of ON... THEN... statements.)

The following program illustrates how the ON SIZE THEN... statement may be used to avoid a SIZE ERROR:

```
100 ON SIZE THEN 220
110 DELETE A,B
120 DIM A(3,4),B(3,4)
130 PRINT "DO YOU WISH TO PERFORM THE INV FUNCTION?"
140 PRINT "IF SO, TYPE '1'; IF NOT, TYPE '0'."
150 INPUT T
160 IF T=0 THEN 250
170 PRINT "ENTER THE ELEMENTS OF A 3X4 MATRIX, IN ROW ORDER:"
180 INPUT A
190 B = INV (A)
200 PRINT " B =INV (A)=";B
210 GO TO 130
220 PRINT "THE VALUE OF THE DERMINANT IS";DET
230 PRINT "THE SQUARE PART OF THE MATRIX HAS NO INVERSE."
240 GO TO 130
250 END
```

This program performs the INV function on 3x4 matrices. The statement ON SIZE THEN 220 alerts the system to the possiblity of a SIZE ERROR occurring at execution time. If a SIZE ERROR occurs while the program is running, control is transferred to line 220. A message then appears on the display, explaining that since the square part of the parameter has no inverse, the INV function cannot be performed. In this way, a SIZE ERROR has been avoided, and the program continues to ask for more data.

Systems of Equations With No Solution

When the square part of the parameter matrix has no inverse and attempting to perform the INV function results in an error, the equations represented by the matrix have no solution.

For example, the square part (the shaded portion) of the matrix shown below has no inverse, and attempting to perform the INV function results in an INVALID FUNCTION ARGUMENT error:

$$C = \begin{bmatrix} 1 & 2 & 1 \\ 3 & 6 & 9 \end{bmatrix}$$

Matrix C represents the following set of two equations in two unknowns:

$$\begin{aligned} x + 2y &= 1 \\ 3x + 6y &= 9 \end{aligned}$$

Since no solution exists for the above equations, it makes sense that the INV function results in an error.

As another example, suppose you need to solve the following equations:

$$\begin{aligned} y + 4z &= 1 \\ -x \qquad + 3z &= 0 \\ 2x + y - 2z &= 1 \end{aligned}$$

The first step is to represent the equations by a matrix:

$$D = \begin{bmatrix} 0 & 1 & 4 & 1 \\ -1 & 0 & 3 & 0 \\ 2 & 1 & -2 & 1 \end{bmatrix}$$

The next step is to perform the INV function on matrix D, in order to obtain a new 3x4 matrix whose fourth column represents the solution for the set of equations.

However, the square part of matrix D has no inverse, and the INV function returns an INVALID FUNCTION ARGUMENT error. The INV function is unable to solve the equations.

As in the previous example, it makes sense that the INV function cannot compute an answer, because no solution exists for the three equations. 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).

Attempting to perform the INV function on a matrix which has more than 255 rows or more than 255 columns causes a SHAPE ERROR.

Supplementary Information

The following paragraphs provide supplementary information about the suitability of certain matrices for inversion and computation of the determinant. The "condition" of a matrix is a very complicated topic, and cannot be fully explained in a few pages. The information is presented as concisely as possible, and is intended only to help you understand how the properties of a particular matrix may affect the results of the INV and DET functions.

Mathematical Invertibility Versus Numerical Invertibility

In mathematical theory, a matrix either has an inverse or does not have an inverse. If a matrix has no inverse, the value of the determinant is zero, and the matrix equation A MPY X = B has no solution (or cannot be uniquely solved). But if a matrix has an inverse, the determinant is non-zero, and A MPY X = B may be solved for any B.

In practice, however, a matrix is called non-invertible when the sequence of arithmetic operations used to compute the inverse reaches a point past which the calculations cannot proceed further. This definition is not the same as the mathematical definition of invertibility. Many matrices are invertible in the mathematical sense, but not in the computational sense. For instance, the following matrix is mathematically invertible, but the inverse cannot necessarily be computed if ε is much smaller than 1:

$$A = \begin{bmatrix} 1 - \varepsilon & 1 + \varepsilon \\ 1 & 1 + 2\varepsilon \end{bmatrix}$$

Although theoretically an inverse exists for this matrix, most machines are unable to compute an answer.

Conversely, a mathematically non-invertible matrix may be "inverted" on most machines without causing an error. For example:

$$B = \begin{bmatrix} a & 1 \\ a\uparrow 2 & a \end{bmatrix}$$

In theory, this matrix has no inverse, yet in practice most algorithms used to compute inverses return an answer if 1/a cannot be represented with complete precision (a=3 is an example of this).

Truncation Error

Truncation is the reason for the difference between what should theoretically happen and what actually happens when a machine is used to calculate an inverse. Truncation can occur within any machine while arithmetic operations are being performed, and while numbers are being converted from decimal to binary representations. Whenever a binary number has more bits than the machine can represent, the extra bits are chopped off (truncated). Truncation is unavoidable, since there are many numbers which cannot be represented with complete precision on any machine.

When a sequence of arithmetic operations is performed, some truncation usually occurs. The total amount of error depends on the numbers involved in the calculations. For matrix inversion, the numbers involved are the values assigned to the elements of the matrix, and the "condition" of a matrix is an indication of how these values influence the total amount of truncation error.

The Condition of a Matrix

A matrix C is called "ill-conditioned" with respect to inversion when small changes in the elements of C cause large changes in the computed inverse of C, or cause C to be non-invertible in a computational sense. It may not be worth attempting to compute the inverse of a very ill-conditioned matrix, because small errors in the original data (the values assigned to the elements of the matrix) can cause the computed answer to be unrecognizably different from the true inverse of the matrix. Ill-conditioned matrices also tend to be more susceptible to truncation error than other matrices.

A matrix which is ill-conditioned with respect to inversion is more susceptible to truncation error in the sense that as arithmetic operations proceed to compute the inverse, truncation error tends to accumulate faster than it would for better-conditioned matrices. Normal amounts of truncation error do "pile up" even for well-conditioned matrices, just because so many computations are needed to invert a matrix, and every step in the process provides another chance for error to occur. (Thus large matrices are more susceptible to truncation error than small ones.) Yet when an ill-conditioned matrix is inverted, much more than the normal amount of truncation error may accumulate, causing the result to be too inaccurate to be called an inverse at all.

The most extreme example of this is a non-invertible matrix, that is, one which has no inverse in the mathematical sense. This type of matrix is the worst case of ill-conditioning with respect to inversion, and if the algorithm used to compute the inverse is able to return an answer, the answer is completely inaccurate. So much truncation error has accumulated during the calculations that the machine appears to have found an inverse, when actually no inverse exists.

The condition of a matrix with respect to inversion can be measured, and the measure is called the condition number k. For any matrix C, the condition number may be calculated as follows:

$$k = |\text{ largest element of C } * \text{ largest element of INV(C) }|$$

If k is close to 1, matrix C is well-conditioned with respect to inversion. However, if k is much larger than 1, C is less well-conditioned; and if k is extremely large, C is called ill-conditioned, and the computed inverse of C is likely to be very far from the true inverse of C.

The number 1/k is a rough measure of the "distance" from matrix C to the nearest mathematically non-invertible matrix. For instance, if the condition number k is 10^{15}, C may be made non-invertible by increasing or decreasing at least one of its elements by about 10^{-15} times the largest element in the matrix.

Evaluating the Result of the INV Function. The condition number may be used as a check of the accuracy of the result of the INV function. If the distance 1/k from C to the closest non-invertible matrix is comparable in magnitude to possible errors in the data (the values assigned to the matrix), the result of performing the INV function may be so inaccurate that INV(C) is meaningless. For example, if the values assigned to the elements of matrix C are physical observations which are only accurate to three places, a condition number k of 10^3 or higher indicates that the result of INV(C) may be too inaccurate to be meaningful. Likewise, if the distance 1/k from C to the closest non-invertible matrix is comparable in magnitude to the precision of the machine, 10^{-15}, the result of INV(C) may be too inaccurate to be of any practical value. This means that if the condition number k is 10^{15}, the distance from C to the closest non-invertible matrix is roughly 10^{-15}, which is the same as the amount of error which might normally occur during any arithmetic operation such as addition or subtraction. In general, if the condition number k is 10^{15} or larger, the matrix may be so ill-conditioned with respect to inversion that the INV function cannot be expected to compute a meaningful result.

Evaluating the Result of the DET Function. The value of the determinant is computed during the INV process. For this reason, the result of the DET function is affected by the condition of the matrix in the same manner as the INV function. When the result of the INV function is inaccurate because the parameter matrix is ill-conditioned with respect to inversion, the value returned by the DET function is not likely to be close to the true value of the determinant.

Thus, if the values assigned to the elements of the matrix are based on physical observations, 1/k should not be smaller than possible errors in the original data. Likewise, a condition number k greater than or equal to 10^{15} indicates that the parameter matrix is so sensitive to truncation error that the result of the DET function may be very different from the true value of the determinant.

This means that the value returned by the DET function cannot be used as an indication of whether or not a matrix is invertible. In mathematical theory, a matrix has no inverse when the value of the determinant is 0; but in practice, if the parameter matrix is non-invertible or ill-conditioned with respect to inversion, the value returned by the DET function may not even be close to 0.

Suppose you want to use the INV function to find the inverse of the following 3x3 matrix:

$$R = \begin{bmatrix} 1 & 0 & 1 \\ 2 & -1 & 5 \\ 3 & -1 & 6 \end{bmatrix}$$

Performing INV(R) results in the following matrix:

$$S=INV(R) = \begin{bmatrix} 6.251999482E+13 & 6.254999482E+13 & -6.254999482E+13 \\ -1.876499845E+14 & -1.876499845E+14 & 1.876499845E+14 \\ -6.254999482E+13 & -6.254999482E+13 & 6.254999482E+13 \end{bmatrix}$$

Since performing the INV function does not result in an error, you might be tempted to accept matrix S as the inverse of matrix R, without checking the accuracy of the result S.

But a check of the condition number k reveals that matrix R is extremely ill-conditioned with respect to inversion:

$$\begin{aligned} k &= |\text{ largest element of } R * \text{ largest element of } S\,| \\ &= R(3,3)*S(2,3) \\ &= 1.125899907\text{ E+15} \end{aligned}$$

The condition number k is so large, and R is so ill-conditioned with respect to inversion, that the INV function cannot be expected to return an accurate inverse.

Since the size of the condition number indicates that R may not be accurately inverted, it is a good idea to check the result by performing R MPY S and S MPY R. How accurate the result returned by the INV function is, depends on how closely the elements of the products resemble those of the 3x3 identity matrix.

The products R MPY S and S MPY R are shown below:

$$M = R \text{ MPY } S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 2 & 2 & 0 \end{bmatrix}, \qquad N = S \text{ MPY } R = \begin{bmatrix} 3 & -0.5 & 2 \\ -6 & 2 & -8 \\ -2 & 0.5 & -2 \end{bmatrix}$$

Neither product resembles the 3x3 identity matrix, confirming that matrix R is too ill-conditioned to be accurately inverted.

In general, after using the INV function to invert a matrix and solve systems of linear equations or after performing the DET function, it is useful to check the condition number k. An extremely large condition number indicates that the parameter matrix is not suitable for matrix inversion or computation of the determinant, and that the results of the INV and DET functions should not be assumed to be accurate.

The MPY Function

The MPY function returns the matrix product of two arrays:

```
[ Line number ] array variable = array variable MPY array variable
```

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.

For instance, to do this matrix multiplication by hand, start with I=1 and J=1 and multiply the elements of the first row of A by the corresponding elements of the first column of B, then add:

$$1 * 2 + 2 * 3 = 8$$

This is the element found in the first row, first column of C:

$$C(1,1) = 8$$

Next, make I=1 and J=2. Multiply each element in the first row of A by the corresponding element in the second column of B, and add:

$$1 * 8 + 2 * 4 = 16$$

This is the value assigned to the first row, second column of C:

$$C(1,2) = 16$$

Now, let I=2 and J=1. Using the second row of A and the first column of B,

$$C(2,1) = 0 * 2 + (-3) * 3 = -9$$

And finally, I=2 and J=2. Using the second row of A and the second column of B,

$$C(2,2) = 0 \cdot 8 + (-3) \cdot 4 = -12$$

These calculations are summarized in the diagram below. At each step, the shaded row of the first matrix and the shaded column of the second matrix are the items used to compute the shaded element of the product matrix:

STEP 1: $\begin{matrix}1 & 2\\ 0 & -3\end{matrix} \; \begin{matrix}2 & 8\\ 3 & 4\end{matrix} = \begin{matrix}8 & \\ & \end{matrix}$, $C(1,1) = \begin{matrix}1 & 2\end{matrix} \; \begin{matrix}2\\ 3\end{matrix} = 1*2+2*3=8$

STEP 2: $\begin{matrix}1 & 2\\ 0 & -3\end{matrix} \; \begin{matrix}2 & 8\\ 3 & 4\end{matrix} = \begin{matrix}8 & 16\\ & \end{matrix}$, $C(1,2) = \begin{matrix}1 & 2\end{matrix} \; \begin{matrix}8\\ 4\end{matrix} = 1*8+2*4=16$

STEP 3: $\begin{matrix}1 & 2\\ 0 & -3\end{matrix} \; \begin{matrix}2 & 8\\ 3 & 4\end{matrix} = \begin{matrix}8 & 16\\ -9 & \end{matrix}$, $C(2,1) = \begin{matrix}0 & -3\end{matrix} \; \begin{matrix}2\\ 3\end{matrix} = 0*2+(-3)*3=-9$

STEP 4: $\begin{matrix}1 & 2\\ 0 & -3\end{matrix} \; \begin{matrix}2 & 8\\ 3 & 4\end{matrix} = \begin{matrix}8 & 16\\ -9 & -12\end{matrix}$, $C(2,2) = \begin{matrix}0 & -3\end{matrix} \; \begin{matrix}8\\ 4\end{matrix} = 0*8+(-3)*4=-12$

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:

$$A = \underbrace{\begin{bmatrix}2 & 3 & -4\\ 3 & -1 & 2\end{bmatrix}}_{2x3} \quad , \; B = \underbrace{\begin{bmatrix}-1 & 3 & 0\\ 2 & 0 & 5\\ 0 & 4 & 1\end{bmatrix}}_{3x3}$$

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.

Performing E MPY F results in an error for the following matrices:

$$E = \underbrace{\begin{bmatrix}1 & 0 & 3 & 0\\ 0 & 1 & 0 & 0\\ -2 & 0 & 2 & 0\\ 0 & 4 & 0 & 1\end{bmatrix}}_{4x4} \quad , \; F = \underbrace{\begin{bmatrix}1 & 0 & 0\\ -1 & 2 & 0\\ 2 & 1 & 1\end{bmatrix}}_{3x3}$$

Here, matrix E has four columns and matrix F has three rows. The product E MPY F does not exist, and performing G = E MPY F results in an error, because the number of columns in E is not the same as the number of rows in F.

The fact that A may be multiplied by B does not necessarily mean that B may be multiplied by A. For instance, in the first example C = A MPY B may be performed, but D = B MPY A results in an error. The product B MPY A does not exist, because the number of columns in B is not equal to the number of rows in A:

$$B = \underset{3x3}{\begin{bmatrix} 1 & -2 & 1 \\ 2 & 1 & -3 \\ 0 & 1 & 1 \end{bmatrix}} \quad , \; A = \underset{2x3}{\begin{bmatrix} 1 & 0 & 2 \\ 2 & -1 & 1 \end{bmatrix}}$$

Since matrix B has three columns and matrix A has two rows, the product B MPY A does not exist.

As a final example, if A is a 2x6 matrix, C = A MPY B may be performed for any matrix B which has six rows. That is B may be a 6x1 matrix, a 6x2 matrix, a 6x3 matrix, or a 6x4 matrix, and so on.

In summary, C = A MPY B can be performed whenever B has as many rows as A has columns.

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:

$$A = \underset{2x3}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}} \quad , \; B = \underset{3x3}{\begin{bmatrix} 3 & 1 & -2 \\ 0 & -1 & 4 \\ 2 & 0 & 0 \end{bmatrix}} \quad , \; C = A\,MPY\,B = \underset{2x3}{\begin{bmatrix} 3 & 1 & -2 \\ 0 & -1 & 4 \end{bmatrix}}$$

Since A has two rows and B has three columns, there are two rows and three columns in the result C.

Here is another example:

$$A = \begin{bmatrix} 3 & 1 \\ -1 & 1 \\ 7 & 0 \end{bmatrix} \quad , B = \begin{bmatrix} 2 & -1 & 0 \\ 1 & 5 & -2 \end{bmatrix}$$

$$C = A \text{ MPY } B = \begin{bmatrix} 3 & 1 \\ -1 & 1 \\ 7 & 0 \end{bmatrix} \begin{bmatrix} 2 & -1 & 0 \\ 1 & 5 & -2 \end{bmatrix} = \begin{bmatrix} 7 & 2 & -2 \\ -1 & 6 & -2 \\ 14 & -7 & 0 \end{bmatrix}$$

$$D = B \text{ MPY } A = \begin{bmatrix} 2 & -1 & 0 \\ 1 & 5 & -2 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ -1 & 1 \\ 7 & 0 \end{bmatrix} = \begin{bmatrix} 7 & 1 \\ -16 & 6 \end{bmatrix}$$

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 A, which means D is a 2x2 matrix:

B MPY A = D

2x3 3x2→2x2

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.

The MPY function returns the matrix product of two arrays. These parameter arrays must previously appear in a DIM statement, using two subscripts to dimension each variable. The dimensions of the parameters must be such that the arrays can be multiplied. That is, in order to perform A MPY B, the second subscript used to dimension A must be the same as the first subscript used to dimension B. For example:

```
100 DIM A(7,3),B(3,12)
```

When the MPY function is performed, a new matrix is generated. The result must be assigned to a target variable. The target 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 following program illustrates how the MPY function can be used to find the matrix product of two arrays:

```
100 DELETE A,B,C
110 DIM A(3,3),B(3,2),C(3,2)
120 READ A,B
130 DATA 1,2,0,4,1,2,-1,3,0,5,0,-6,1,2,0
140 C = A MPY B
150 PRINT "A=";A
160 PRINT "B=";B
170 PRINT "C = A MPY B=";C
```

This program performs matrix multiplication on two arrays A and B, and prints the result. The parameter matrices A and B are dimensioned in line 110. Since A is dimensioned to be a 3x3 matrix, and B is dimensioned to be a 3x2 matrix, the result of A MPY B is dimensioned in the same DIM statement to be a 3x2 matrix called C. When line 140 is executed, A MPY B is performed and the result is assigned to array variable C. The next three statements cause the results to appear on the display, as indicated here:

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 4 & 1 & 2 \\ -1 & 3 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 0 \\ -6 & 1 \\ 2 & 0 \end{bmatrix}, \quad C = A \text{ MPY } B = \begin{bmatrix} -7 & 2 \\ 18 & 1 \\ -23 & 3 \end{bmatrix}$$

The target for the result of the MPY function must not have the same variable name as either of the parameters. That is, the following types of statements are not valid:

```
A = A MPY B
B = A MPY B
```

Using the same variable name for the target array as for one of the parameters results in an error.

Attempting to perform the MPY function without assigning a numeric value to every element of both parameters causes an UNDEFINED VARIABLE error message to be printed on the display.

Attempting to perform the MPY function causes a SIZE ERROR when one or more of the elements in the result is out of range (outside the range ±1.0E±308). After a SIZE ERROR message appears on the display, the result of performing the MPY function may be obtained by entering the target variable name and pressing RETURN. Elements in the result which are out of range (too large or too small) contain the largest or smallest number possible, ±8.988465774E+307.

A MPY B vs B MPY A

Matrix multiplication is not commutative, that is, A MPY B does not generally return the same result as B MPY A. For instance:

$$A = \begin{bmatrix} 0 & 1 \\ 2 & -5 \end{bmatrix} \quad , \quad B = \begin{bmatrix} 4 & 3 \\ 0 & 2 \end{bmatrix} \quad ,$$

$$C = \text{A MPY B} = \begin{bmatrix} 0 & 1 \\ 2 & -5 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 8 & -4 \end{bmatrix}$$

$$D = \text{B MPY A} = \begin{bmatrix} 4 & 3 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & -5 \end{bmatrix} = \begin{bmatrix} 6 & -11 \\ 4 & -10 \end{bmatrix}$$

Notice that matrices C = A MPY B and D = B MPY A are not the same. This is to be expected when performing matrix multiplication.

Since the results of A MPY B and B MPY A are usually different, care must be taken when determining which of the two products A MPY B or B MPY A is needed. Making the wrong choice may result in an incorrect answer, or even an error message: in many cases it is possible to perform A MPY B, but not B MPY A because of mismatched dimensions.

The Zero Matrix

One unusual property of matrix multiplication concerns the matrix called 0 because it is filled entirely with 0's. In ordinary arithmetic, X * Y = 0 means that either X or Y or both are 0. But this is not the case in matrix algebra: A MPY B may be 0 without either A or B being 0. Here is an example:

$$A = \begin{bmatrix} 1 & -1 \\ 2 & -2 \end{bmatrix} \; , \; B = \begin{bmatrix} 1 & 3 \\ 1 & 3 \end{bmatrix} \; , \; \text{A MPY B} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Here, neither A nor B is 0, yet the product A MPY B is 0. Keep in mind that this can happen when using the MPY function, and don't be alarmed when it does.

The Difference Between * and the MPY Function

There is more than one way to multiply matrices. One way uses the arithmetic operator · to form the element-by-element product of matrices A and B. The other way uses the MPY function to form the "matrix product" of A and B. The matrix product and the element-by-element product are computed in different ways, and yield different answers. For example:

$$A = \begin{bmatrix} 1 & 2 \\ 0 & -3 \end{bmatrix} , \quad B = \begin{bmatrix} 2 & 8 \\ 3 & 4 \end{bmatrix} \quad ,$$

$$C = \quad A * B = \begin{bmatrix} 1 & 2 \\ 0 & -3 \end{bmatrix} \begin{bmatrix} 2 & 8 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 16 \\ 0 & -12 \end{bmatrix}$$

$$D = \text{A MPY B} = \begin{bmatrix} 1 & 2 \\ 0 & -3 \end{bmatrix} \begin{bmatrix} 2 & 8 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 8 & 16 \\ -9 & -12 \end{bmatrix}$$

Notice that A * B and A MPY B give different results. A * B is called the element-by-element product of A and B, because each element in the result is the product of the corresponding elements of A and B. That is, C = A * B is calculated as follows: C(1,1) = A(1,1) * B(1,1) = 1 * 2 = 2, C(1,2) = A(1,2) * B(1,2) = 2 * 8 = 16, and so on. However, D = A MPY B, the matrix product of A and B, is obtained in a different manner.

Do not confuse the operation A * B with the matrix product provided by the MPY function. Matrix multiplication MPY has a special relationship to systems of linear equations, and has many other significant applications. You will find when working with matrices that the words "product" and "multiplication" often mean the **matrix** product and **matrix** multiplication, as provided by the MPY function. In such situations, using the operator * instead of the MPY function results in incorrect answers.

Raising Matrices to a Power

A matrix may be raised to a power in more than one way. One method uses the operator ↑ or * to raise each element of an array to a specified power. Another method finds the powers of a matrix by using the MPY function.

For example, A↑2 creates a matrix whose elements are the square of the corresponding elements in A. Likewise, A · A is the element-by-element product of A with itself, and causes each element of A to be squared in the same manner as A↑2. However, A MPY A squares A by computing the matrix product of A with itself, and gives a different result, as seen in the following example:

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \quad , \quad A{*}A{=}A{\uparrow}2{=} \begin{bmatrix} 0 & 1 \\ 4 & 9 \end{bmatrix} \quad , \quad B{=}A \text{ MPY } A{=} \begin{bmatrix} 2 & 3 \\ 6 & 11 \end{bmatrix}$$

It is important not to confuse these methods of raising matrices to a power. Certain situations call for the "power" of a matrix to be computed by repeatedly performing the MPY function. In such cases, using the operator ↑ or * by mistake produces incorrect answers.

How Matrix Multiplication Relates to Systems of Linear Equations

A set of simultaneous equations can be represented by a matrix product. For example, consider the following equations:

$$3x + 2y = 5$$
$$4x + y = 11$$

An equivalent way to state the equations uses a matrix product:

$$\begin{bmatrix} 3 & 2 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$$

The first matrix in the above expression is called the "coefficient matrix" because its elements are the coefficients of x and y in the original set of equations. The second matrix is a column consisting of the unknowns x and y, and the third matrix is a column consisting of the constants which appear to the right in the original set of equations.

If A is the coefficient matrix, X is the column of unknowns, and B is the column of constants, the above expression can be written more compactly as follows:

$$AX = B$$

AX is the matrix product of A and X. The expression AX = B is equivalent to the original set of equations. This fact may be verified by working out the matrix multiplication as follows:

$$AX = \begin{bmatrix} 3 & 2 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3x + 2y \\ 4x + \ \ y \end{bmatrix} = B$$

But, in this example, B is the column $\begin{bmatrix} 5 \\ 11 \end{bmatrix}$, therefore the equation AX = B says the following:

$3x + 2y = 5$
and
$4x + y = 11$

These are the original two equations. In other words, the matrix product form is an equivalent way to state a system of equations.

NOTE

Do not enter a matrix of the type $\begin{bmatrix} x \\ y \end{bmatrix}$ from the keyboard. The elements of a matrix must always be assigned numeric values, and attempting to assign variable names such as x or y result in an error. The discussion above is provided only to help you understand the relationship between linear equations and matrix multiplication.

The TRN Function

The TRN function returns the transpose of a matrix:

```
[ Line number ]  array variable = TRN  array variable
```

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:

$$A = \begin{bmatrix} 1 & 3 & 5 \\ 8 & -2 & 0 \end{bmatrix} \quad , B = \text{TRN (A)} = \begin{bmatrix} 1 & 8 \\ 3 & -2 \\ 5 & 0 \end{bmatrix}$$

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:

$$A = \begin{bmatrix} 6 & 1 & 8 \end{bmatrix} \quad , B = \text{TRN (A)} = \begin{bmatrix} 6 \\ 1 \\ 8 \end{bmatrix}$$

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. Not assigning a numeric value to one or more of the elements of the parameter matrix, causes an UNDEFINED VARIABLE error message to be printed on the display when the TRN function is performed.

If a matrix is to be used as a parameter for the TRN function, it must be dimensioned in a DIM statement, using two subscripts to indicate the number of rows and columns in the matrix. For example, if a 5x6 matrix A is to be supplied to the TRN function, A must be dimensioned in the following statement:

```
DIM A(5,6)
```

When the parameter for the TRN function is a matrix consisting of one row of four elements, the matrix should not be dimensioned as DIM A(4). The correct way to dimension the matrix is:

```
DIM A(1,4)
```

When the TRN function is performed, the result is a new matrix, which must be assigned to an array variable, called the target variable because it serves as a "target" for the new matrix. The target 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. Forgetting to dimension the target variable causes an INVALID FUNCTION ARGUMENT message to appear on the display.

The following program illustrates how the TRN function may be used to find the transpose of a matrix:

```
100 DELETE A,B
110 DIM A(2,3),B(3,2)
120 READ A
130 DATA 1,3,5,8,−2,0
140 B = TRN (A)
150 PRINT B
```

This program computes the transpose of a 2x3 matrix A, and assigns the result to target variable B. The first line of the program deletes variables A and B from memory. This is a precautionary measure to make sure that the variables are in an undefined state before line 110 is executed. (Previously defined numeric variables cannot be dimensioned as array variables without first deleting them from memory.)

Line 110 dimensions the variable A to be a 2x3 matrix. Variable B is to be used as the target variable, and is dimensioned in the same statement to be a 3x2 matrix.

When line 120 is executed, the numbers contained in line 130 are assigned to matrix A in row major order. The result is shown below:

$$A = \begin{bmatrix} 1 & 3 & 5 \\ 8 & -2 & 0 \end{bmatrix}$$

In line 140, TRN(A) is performed and the result is assigned to the target variable. Finally, line 150 causes the result to appear on the display. The result is shown below:

$$B = TRN\ (A) = \begin{bmatrix} 1 & 8 \\ 3 & -2 \\ 5 & 0 \end{bmatrix}$$

The transpose of a matrix A does not have the same shape as matrix A, unless A is square (has the same number of rows as columns). For this reason, caution must be used when giving the same name to the target variable as to the parameter variable. For example:

```
100 DELETE A
110 DIM A(5,2)
120 READ A
130 DATA 1,2,6,−4,3,0,8,1,−2
140 A = TRN (A)
150 PRINT A
```

This program causes a SHAPE ERROR message to appear on the display. The SHAPE ERROR occurs when line 140 is executed, because the statement 140 A=TRN(A) uses the same variable name for the target variable and the parameter of the TRN function. When the statement is executed, an attempt is made to assign the transpose of A, a 2x5 matrix, to array variable A, which is dimensioned in line 110 to be a 5x2 matrix. A SHAPE ERROR results.

When matrix A is square (has the same number of rows as columns), the transpose of A **does** have the same shape as A. In this case, the same variable name can be used for the target variable as for the parameter of the TRN function. For example:

```
100 DELETE A
110 DIM A(3,3)
120 READ A
130 DATA 1,2,6,−4,3,0,8,1,−2
140 A = TRN (A)
150 PRINT A
```

This program is the same as the previous one, but lines 100 and 120 have been modified to make A a square 3x3 matrix. This time, line 140 does not cause a SHAPE ERROR, because A and the transpose of A are of the same shape and have the same number of elements. When the statement A = TRN(A) is executed, the transpose of A is computed and assigned to array variable A, replacing the original matrix A. The statement 140 PRINT A; causes the result to appear on the display as indicated below:

$$A = \begin{bmatrix} 1 & -4 & 8 \\ 2 & 3 & 1 \\ 6 & 0 & -2 \end{bmatrix}$$

The statement A = TRN(A) causes the original matrix A to be replaced by TRN(A). Care must be taken not to "overwrite" the original matrix A in this way, if A is needed in later parts of the program. In general, it is safer to use different variable names for the target variable and the parameter variable; however, if a matrix has been overwritten by its transpose, the original matrix can be recovered by performing the TRN function again.

SUMMARY

An array is an ordered collection of data items arranged in a defined pattern. Arrays in the Graphic System can be one or two dimensional. Elements of one dimensional arrays are referenced with a single subscript; elements of two dimensional arrays are referenced with two subscripts.

The memory required to contain an array is allocated with the DIM statement. You can dimension an array to a smaller number of elements with no problem, but to re-dimension an array to a larger number of elements, you must first delete the array from memory and then re-allocate the space.

You can input data to an array without having to utilize a FOR/NEXT loop for subscripts by using INPUT or READ. The array is filled in row major order, i.e., the first row is filled first from the lowest-to-highest subscript, then the second row, and so on. Data can be output similarly, using PRINT. Array assignments can be used to copy one array into another, or to set all the elements of an array equal to a single value. You can perform arithmetic operations on arrays just as you would with numeric variables.

In addition to performing simple arithmetic operations on arrays, the Graphic System performs algebraic operations on matrices. These algebraic operations are called matrix functions to distinguish them from other array operations. The matrix functions provide the determinant (DET), the identity matrix (IDN), the inverse and solutions to systems of linear equations (INV), the matrix product (MPY), and the transpose (TRN).

EXAMPLE PROGRAMS

TITLE: Student Grade Averages

DESCRIPTION: This program inputs student names and grades, and outputs the averages of each student plus the class average. You supply the program with the number of students and the number of grades per student.

PROGRAM LISTING:

```
100 REMARK  STUDENT GRADE AVERAGES
110 PAGE
120 PRINT "ENTER NUMBER OF STUDENTS: ";
130 INPUT N
140 PRINT "ENTER NUMBER OF GRADES PER STUDENT: ";
150 INPUT M
160 DELETE G
170 DIM G(M)
180 PRINT "ENTER STUDENT NAME FOLLOWED BY GRADES"
190 PRINT
200 PRINT
210 C=0
220 FOR J=1 TO N
230 INPUT S$
240 S=0
250 FOR I=1 TO M
260 PRINT "(";I;")  ";
270 INPUT G(I)
280 S=S+G(I)
290 NEXT I
300 A=S/M
310 PRINT "*** AVERAGE= ";A
320 PRINT
330 PRINT
340 C=C+A
350 NEXT J
360 A=C/N
370 PRINT
380 PRINT
390 PRINT "CLASS AVERAGE= ";A
400 END
```

METHODOLOGY: The data is input with a nested FOR loop which also provides visual organization of the data. The DELETE in line 160 permits array "G" to be redimensioned to a larger size on succeeding runs of the program.

OPERATING PROCEDURE: Type RUN, press RETURN. The program responds with ENTER NUMBER OF STUDENTS. Enter the number and press RETURN. The next prompting is ENTER NUMBER OF GRADES PER STUDENT. Enter this number, then press RETURN. The last prompting is ENTER STUDENT NAME FOLLOWED BY GRADES. Enter the name of the first student, press RETURN, then enter the student's grades, separating the grades with RETURN. When the last grade has been entered, the student's average is output. You then enter the name of the next student, etc. When all the information has been entered, the class average is output.

OUTPUT SAMPLE:

```
ENTER NUMBER OF STUDENTS: 3
ENTER NUMBER OF GRADES PER STUDENT: 3
ENTER STUDENT NAME FOLLOWED BY GRADES


STUDENT1
(1)  78
(2)  87
(3)  89
*** AVERAGE= 84.6666666667

STUDENT2
(1)  75
(2)  93
(3)  84
*** AVERAGE= 84

STUDENT3
(1)  91
(2)  89
(3)  90
*** AVERAGE= 90


CLASS AVERAGE= 86.2222222222
```

TITLE: Security Analysis Program

DESCRIPTION: The program inputs monthly closing prices for stocks over a period of time, and outputs the average price and standard deviation for each security. Theoretically, the stock with the largest standard deviation is the one exhibiting the most rapid change in closing prices.

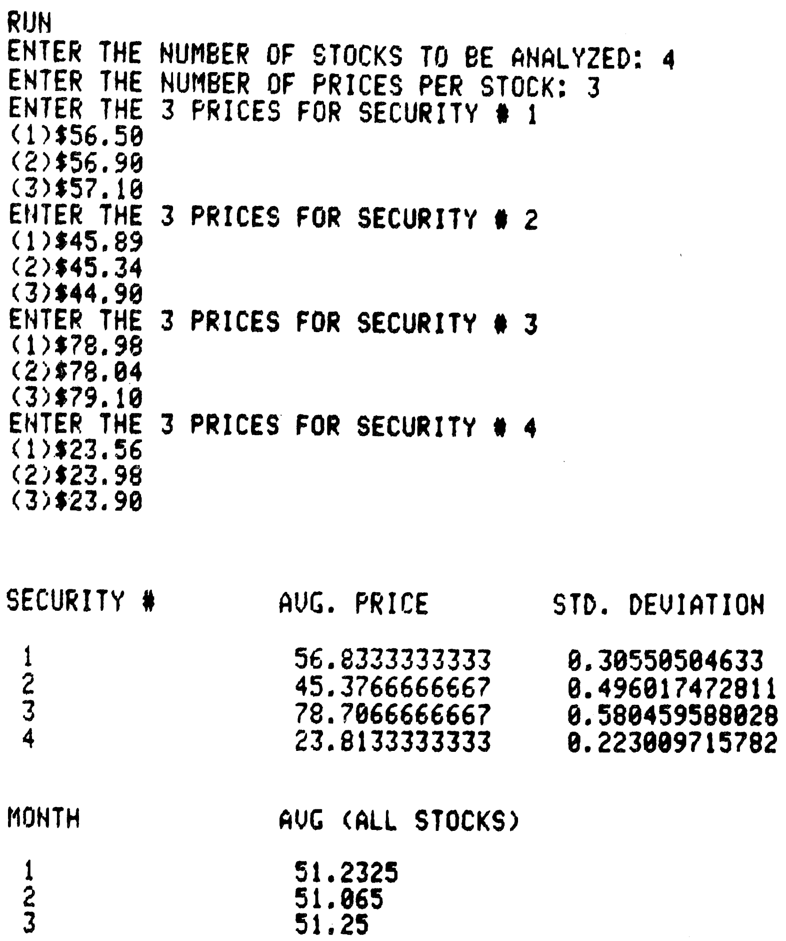
PROGRAM LISTING:

```
100 REMARK ** SECURITY ANALYSIS PROGRAM
110 PRINT "ENTER THE NUMBER OF STOCKS TO BE ANALYZED: ";
120 INPUT N
130 PRINT "ENTER THE NUMBER OF PRICES PER STOCK: ";
140 INPUT M
150 DELETE S
160 DIM S(N,M)
170 REMARK INPUT DATA
180 FOR I=1 TO N
190 PRINT "ENTER THE ";M;" PRICES FOR SECURITY # ";I
200 FOR J=1 TO M
210 PRINT "(";J;")$";
220 INPUT S(I,J)
230 NEXT J
240 NEXT I
250 REMARK OUTPUT THE AVG. PRICE AND STD. DEVIATION
260 PAGE
270 PRINT "SECURITY #","AVG. PRICE","STD. DEVIATION"
280 PRINT
290 FOR I=1 TO N
300 T1=0
310 FOR J=1 TO M
320 T1=T1+S(I,J)
330 NEXT J
340 A=T1/M
350 T2=0
360 FOR J=1 TO M
370 T2=T2+(S(I,J)-A)↑2
380 NEXT J
390 D=(T2/(M-1))↑0.5
400 PRINT I,A,D
410 NEXT I
420 PRINT
430 PRINT
440 REMARK OUTPUT AVERAGES
450 PRINT "MONTH","AVG (ALL STOCKS)"
460 PRINT
470 FOR J=1 TO M
480 T1=0
490 FOR I=1 TO N
500 T1=T1+S(I,J)
510 NEXT I
520 PRINT J,T1/N
530 NEXT J
540 END
```

METHODOLOGY: Nested FOR loops are used to input and output data. The DELETE in line 150 permits upward redimensioning of the array "S". The standard deviation is obtained in line 390.

OPERATING PROCEDURE: Type RUN and press RETURN. The program responds with "ENTER THE NUMBER OF STOCKS TO BE ANALYZED". Enter the appropriate figure and press RETURN. The next prompt is "ENTER THE NUMBER OF PRICES PER STOCK". Enter a number corresponding to the number of closing prices involved, and press RETURN. The program then goes on to ask you to enter the various prices. Once this is done, the average price and standard deviation for each set of stocks is output, followed by the average price for all stocks for each month.

OUTPUT SAMPLE:



```
RUN
ENTER THE NUMBER OF STOCKS TO BE ANALYZED: 4
ENTER THE NUMBER OF PRICES PER STOCK: 3
ENTER THE 3 PRICES FOR SECURITY # 1
(1)$56.50
(2)$56.90
(3)$57.10
ENTER THE 3 PRICES FOR SECURITY # 2
(1)$45.89
(2)$45.34
(3)$44.90
ENTER THE 3 PRICES FOR SECURITY # 3
(1)$78.98
(2)$78.04
(3)$79.10
ENTER THE 3 PRICES FOR SECURITY # 4
(1)$23.56
(2)$23.98
(3)$23.90


SECURITY #          AVG. PRICE         STD. DEVIATION

 1                   56.8333333333      0.30550504633
 2                   45.3766666667      0.496017472811
 3                   78.7066666667      0.580459588028
 4                   23.8133333333      0.223009715782

MONTH               AVG (ALL STOCKS)

 1                   51.2325
 2                   51.065
 3                   51.25
```

TITLE: **Questionnaire Analysis**

DESCRIPTION: The program inputs data obtained from questionnaires, and outputs a tabulation of the responses. The program also outputs a cross tabulation; that is, it reveals the number of respondents giving each possible combination of responses to any two questions of your choosing. Ten possible responses are allowed for each question. A response of "0" means "no response"; the remaining nine responses can be 1 through 9.

PROGRAM LISTING:

```
100 REM***QUESTIONNAIRE ANALYSIS PROGRAM
110 PAGE
120 PRINT "ENTER NUMBER OF QUESTIONS ON QUESTIONNAIRE ";
130 INPUT N1
140 PRINT "ENTER NUMBER OF CROSS TABULATIONS: ";
150 INPUT N2
160 IF N2>0 THEN 190
170 PRINT "INVALID ENTRY!"
180 GO TO 140
190 DELETE R,X,Y,C,T
200 DIM R(N1),X(N2),Y(N2),C(N2,100),T(N1,10)
210 T=0
220 C=0
230 N3=0
240 PRINT "ENTER QUESTIONS TO BE CROSS-TABULATED (N,N): "
250 FOR I=1 TO N2
260 INPUT X(I),Y(I)
270 IF X(I)=Y(I) THEN 240
280 NEXT I
290 PRINT "ENTER RESPONSES TO QUESTIONNAIRE NUMBER ";N3+1
300 FOR I=1 TO N1
310 PRINT "(";I;")  ";
320 INPUT R(I)
330 IF R(I)>9 THEN 350
340 IF R(I)=>0 THEN 380
350 PRINT "RESPONSE MUST BE BETWEEN 0 AND 9"
360 PRINT "RE-ENTER RESPONSE (";I;") "
370 GO TO 320
380 T(I,R(I)+1)=T(I,R(I)+1)+1
390 NEXT I
400 REM****SUM CROSS-TABS
410 FOR I=1 TO N2
420 R9=10*R(X(I))+R(Y(I))
430 C(I,R9)=C(I,R9)+1
440 NEXT I
450 N3=N3+1
460 PRINT "ANOTHER QUESTIONNAIRE ? (YES=1, NO=2) ";
470 INPUT Q
480 GO TO Q OF 290,490
490 PAGE
500 REM***OUTPUT RESULTS
```

```
510 PRINT "NUMBER OF QUESTIONNAIRES= ";N3
520 PRINT
530 FOR I=1 TO N1
540 PRINT "RESPONSES TO QUESTION ";I
550 PRINT
560 PRINT "RESPONSE","NUMBER","PERCENT"
570 PRINT "*********************************************"
580 IF T(I,1)=0 THEN 610
590 P=0.1*INT(1000*T(I,1)/N3+0.5)
600 PRINT "NONE",T(I,1),P
610 FOR J=2 TO 10
620 T9=T(I,J)
630 IF T9=0 THEN 660
640 P=0.1*INT(1000*T9/N3+0.5)
650 PRINT J-1,T9,P
660 NEXT J
670 PRINT
680 PRINT
690 NEXT I
700 REM***OUTPUT CROSS-TABS
710 FOR I=1 TO N2
720 PRINT
730 PRINT "CROSS-TABULATION ";I
740 PRINT
750 PRINT "RESPONSE TO","RESPONSE TO"
760 PRINT "QUESTION ";X(I),"QUESTION ";Y(I),"NUMBER","PCT."
770 PRINT "************************************************************"
780 FOR J=1 TO 100
790 C9=C(I,J)
800 IF C9=0 THEN 930
810 R1=INT(J/10)
820 R2=J-10*R1
830 IF R1=0 THEN 860
840 PRINT R1,"";
850 GO TO 870
860 PRINT "NONE","";
870 IF R2=0 THEN 900
880 PRINT R2,"";
890 GO TO 910
900 PRINT "NONE","";
910 P9=0.1*INT(1000*C9/N3+0.5)
920 PRINT C9,P9
930 NEXT J
940 NEXT I
950 PRINT "END OF PROGRAM"
960 END
```

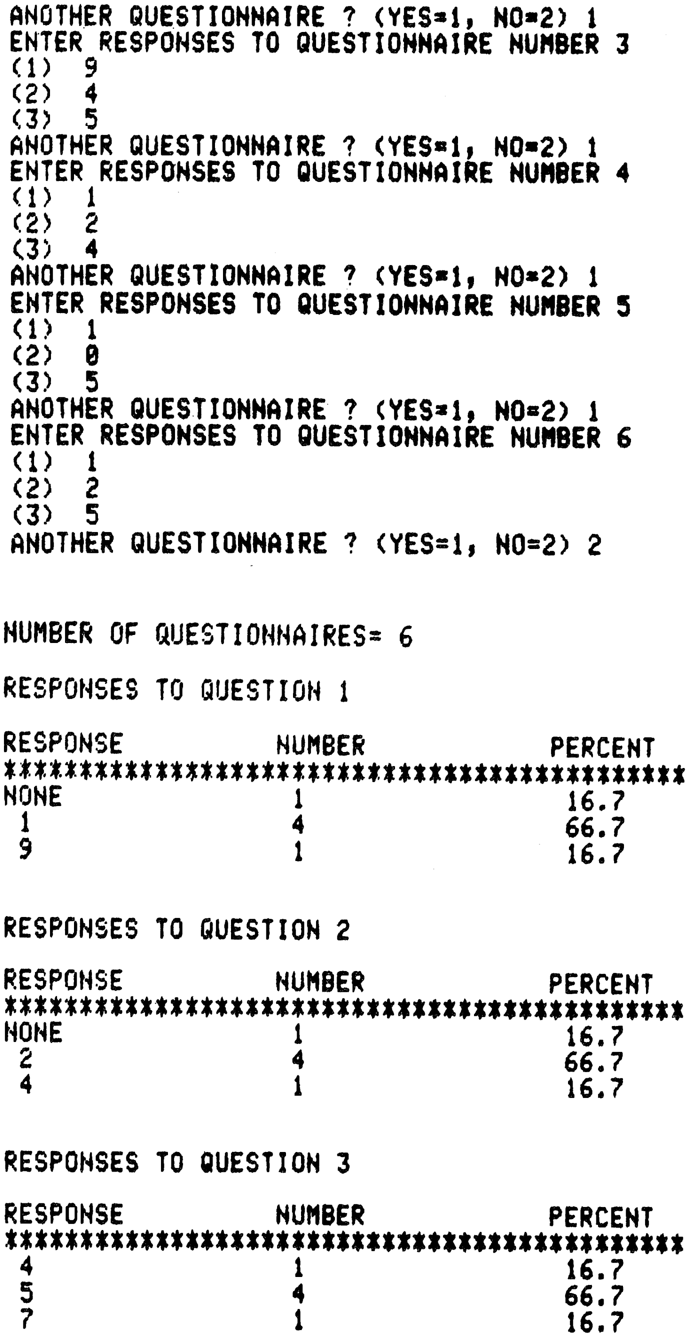
METHODOLOGY: The program involves fairly extensive manipulations with arrays, using the contents of one array to subscript another array. The responses to the questionnaire are stored in array R, and the response totals are stored in array T. The results of the cross-tabulations are accumulated in array C. Array C contains one row for each requested cross-tabulation, and one hundred columns. The one hundred columns in C exist to provide a location corresponding to each possible pair of responses in the cross-tabulation. For example, in cross-tabulation 3, suppose that the response to the "first" question is 9, and the response to the "second" question is 2. In this case, 1 is added to the contents of C(3,92).

The response to the "first" question is multiplied by 10 and summed with the response to the "second" question; this becomes the "column" subscript for array C. See statement 420. Later, this technique is reversed: the question numbers are extracted from the "column" subscript (statements 800 and 810), in effect, the technique makes it possible to simulate a three-dimensional array.

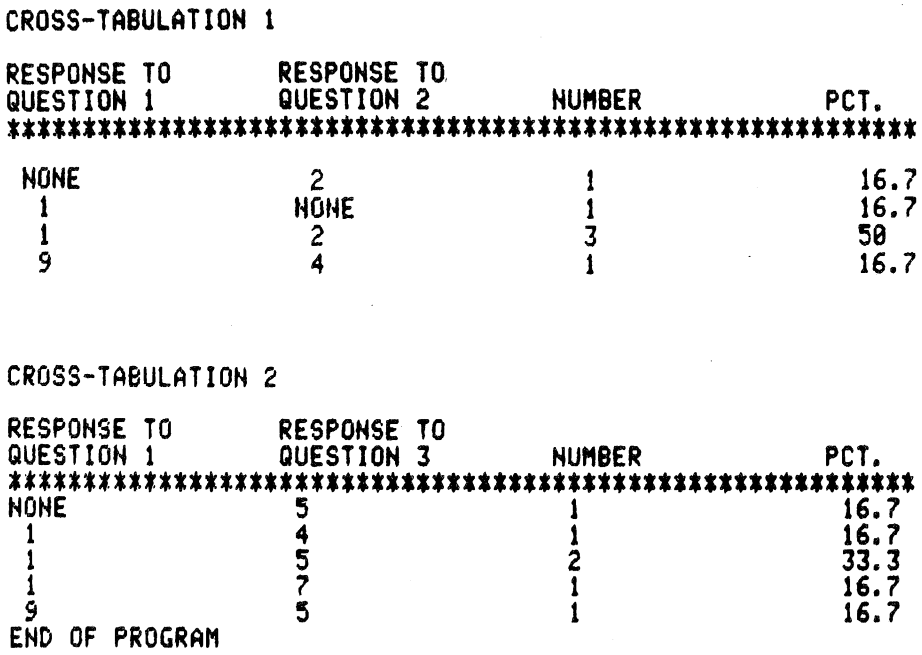
OPERATING PROCEDURE: Type RUN and press RETURN. The program responds with "ENTER NUMBER OF QUESTIONS ON QUESTIONNAIRE". Enter the appropriate number and press RETURN. The program then asks "ENTER NUMBER OF CROSS TABULATIONS". Enter the desired number (≥1) and press RETURN. The next prompting is "ENTER QUESTIONS TO BE CROSS-TABULATED (N,N)". To cross-tabulate questions 1 and 4, for example, enter 1,4. The program will input as many of these pairs as there are questions to be cross-tabulated. (You cannot cross-tabulate a question with itself.) Following this, the program responds with "ENTER RESPONSES TO QUESTIONNAIRE NUMBER 1". Once you have entered this information, following each entry with RETURN, the program asks "ANOTHER QUESTIONNAIRE? (YES = 1, NO = 2)". Enter the appropriate response, and the program continues in this fashion until all the questionnaire data has been input. Following this, the tabulation and requested cross-tabulations are output.

OUTPUT SAMPLE:

```
ENTER NUMBER OF QUESTIONS ON QUESTIONNAIRE 3
ENTER NUMBER OF CROSS TABULATIONS: 2
ENTER QUESTIONS TO BE CROSS-TABULATED (N,N):
1,2
1,3
ENTER RESPONSES TO QUESTIONNAIRE NUMBER 1
(1)  1
(2)  2
(3)  7
ANOTHER QUESTIONNAIRE ? (YES=1, NO=2) 1
ENTER RESPONSES TO QUESTIONNAIRE NUMBER 2
(1)  0
(2)  2
(3)  5
```



```
ANOTHER QUESTIONNAIRE ? (YES=1, NO=2) 1
ENTER RESPONSES TO QUESTIONNAIRE NUMBER 3
(1)  9
(2)  4
(3)  5
ANOTHER QUESTIONNAIRE ? (YES=1, NO=2) 1
ENTER RESPONSES TO QUESTIONNAIRE NUMBER 4
(1)  1
(2)  2
(3)  4
ANOTHER QUESTIONNAIRE ? (YES=1, NO=2) 1
ENTER RESPONSES TO QUESTIONNAIRE NUMBER 5
(1)  1
(2)  0
(3)  5
ANOTHER QUESTIONNAIRE ? (YES=1, NO=2) 1
ENTER RESPONSES TO QUESTIONNAIRE NUMBER 6
(1)  1
(2)  2
(3)  5
ANOTHER QUESTIONNAIRE ? (YES=1, NO=2) 2


NUMBER OF QUESTIONNAIRES= 6

RESPONSES TO QUESTION 1

RESPONSE          NUMBER            PERCENT
*********************************************
NONE               1                 16.7
 1                 4                 66.7
 9                 1                 16.7


RESPONSES TO QUESTION 2

RESPONSE          NUMBER            PERCENT
*********************************************
NONE               1                 16.7
 2                 4                 66.7
 4                 1                 16.7


RESPONSES TO QUESTION 3

RESPONSE          NUMBER            PERCENT
*********************************************
 4                 1                 16.7
 5                 4                 66.7
 7                 1                 16.7
```



```
CROSS-TABULATION 1

RESPONSE TO       RESPONSE TO
QUESTION 1        QUESTION 2        NUMBER            PCT.
*************************************************************

 NONE                2                 1                 16.7
  1                 NONE               1                 16.7
  1                  2                 3                 50
  9                  4                 1                 16.7


CROSS-TABULATION 2

RESPONSE TO       RESPONSE TO
QUESTION 1        QUESTION 3        NUMBER            PCT.
*************************************************************
NONE                5                 1                 16.7
 1                  4                 1                 16.7
 1                  5                 2                 33.3
 1                  7                 1                 16.7
 9                  5                 1                 16.7
END OF PROGRAM
```

Section 5

CHARACTER STRINGS

BACKGROUND

All of the programming techniques discussed so far have been concerned with arithmetic data — numbers have been read into the memory, something is done to the numbers, and numbers appear as output. In some of the examples, alphabetic data has been carried along in the form of literal strings used to enhance output or prompt for input. Up to now, however, no operations have been performed on alphabetic data, or "character strings".

BASIC provides a number of built-in facilities for processing character strings or non-numeric data. This section of the manual addresses the types of things that can be done with strings.

One characteristic that every string has is its length. Length corresponds to the number of characters present in the string. For example, the string "PROGRAM" has a length of seven. If can be envisioned as appearing in the memory like this:

character position	(1)	(2)	(3)	(4)	(5)	(6)	(7)
	P	R	O	G	R	A	M

This brings out another characteristic of strings: each character has its own unique position within the string. The character positions are numbered from left to right, with the number 1 corresponding to the leftmost character.

At this point, some mention should be made concerning the representation of string data in the Graphic System memory. Obviously, if the character string "PROGRAM". . . stored in memory, it must be represented with some numeric equivalent. The machine works with binary numbers (1's and 0's), not with A's, B's, and C's. Also, if each computing system represented string data differently, the programmer's task would approach true chaos instead of its present state of occasional perplexity. To escape this trap, a standardized code has been established that had its beginning with the teletype; the code is the ASCII code (for American Standard Code for Information Interchange). Each character has an equivalent decimal number ranging from 0 to 127. (A tabulation of the ASCII character set is included in the Appendix.) The letter "C", for example, appears in the ASCII code as the binary pattern 1000011, which also is the binary equivalent for the decimal number 67. From the standpoint of programming the Graphic System, all you need be concerned with is the fact that each character has a decimal equivalent. These numbers are within the range of 0 through 127.

String Assignment Statements

String data can be assigned to string variables with the familiar assignment operator. String variables, you will recall, consist of one letter (A through Z) followed by a dollar sign ($), as in A$, B$, X$, etc. String assignment examples:

```
100 LET A$="ABCDE"
110 B$="PROGRAM"
120 C$=A$
```

Generally, string assignment statements look like

[line number] target string = source string

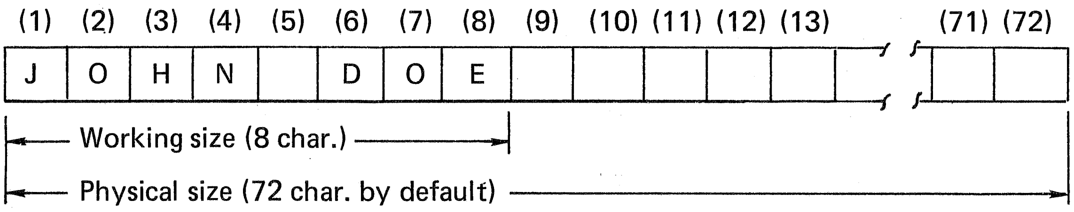
which means that characters are transferred from the source string to the target string. The source string can be a previously defined string variable (as in line 120 above) or a character string enclosed within quotes (lines 100 and 110 above).

Dimensioning String Variables

String variables in the Graphics System have a default length of 72 characters. This corresponds to the maximum number of characters that can be printed on one line of the display. The total length of a string variable can be thought to consist of a working size and a physical size. To illustrate, consider a statement like

```
500 LET A$="JOHN DOE"
```

where A$ has not been previously dimensioned. This commits space in the memory as shown below:



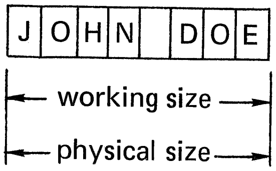
Assuming for the moment that no further characters are to be added to A$, the resulting situation is that a significant amount of space in memory is left unused. The way around this is to allocate space using a DIM statement. The form is essentially the same as the DIM statement used with numeric arrays:

```
[Line number] DIM string variable (numeric expression)
```

where the *numeric expression* is rounded to an integer and becomes the physical size of the string. Strings may be dimensioned longer than 72 characters; the maximum length is bounded only by the amount of memory available. Continuing with the example of "JOHN DOE", a more appropriate utilization of memory results through the following program segment:

```
500 DIM A$(8)
510 LET A$="JOHN DOE"
```

This allocates memory for A$ as follows:



Here, the working size and the physical size are the same, resulting in a more efficient use of memory space.

If you attempt to assign a source string to a target having insufficient space to contain it, an error will result. That is, if you attempt to place a ten character string into a string variable that has been dimensioned to contain five characters, you will receive an error message in return.

String Input/Output

The INPUT, READ/DATA, and PRINT statements can be used with string data in essentially the same manner as with numeric data. A statement like

```
150 INPUT A$
```

behaves as before, placing a question mark on the screen indicating that the BASIC interpreter is awaiting data. You then enter the appropriate character string and press RETURN. The string need not be enclosed in quotes. If the string is enclosed in quotes, then the quote marks just become part of the string.

Similarly, the READ statement causes the BASIC interpreter to assign the next data item in a DATA statement to the variable specified in the READ statement, as in

```
200 READ A$, B$
    .
    .
    .
    .
500 DATA "JOHN DOE", "1530 MAIN ST."
```

String values in DATA statements must be enclosed in quotes and separated by commas. Notice in statement 500 above the presence of digits in a character string. It is important to recognize that a distinction exists between the number 5 and the character 5. Numbers can enter into arithmetic operations; characters are ASCII characters like "X", "Y", "7", and "#". The letter "4", for example, cannot be used in a math operation.

The fact that character strings enclosed within quotes can appear in a DATA statement suggests that you can mix numeric and non-numeric data in a DATA statement, and indeed you can. For instance:

```
150 READ X, A$
160 READ Y, Z
    .
    .
    .
    .
700 DATA 214, "JONES", 5.75, 40
```

Obviously, you have to ensure that the order of the data types appearing in the READ statement corresponds to the data present in the DATA statement. If you attempt to READ string data, but the next data item in the DATA statement is numeric (not enclosed in quotes), you will be presented with an error message. Similarly, if you try to READ numeric data, and the next data item is non-numeric, an error will occur.

String variables can also be mixed with numeric variables in a PRINT statement, as in

```
200 PRINT X$;Y(N);C$;Y(N+1)
```

String Comparisons

Control of program execution can be accomplished through string comparisons resulting in conditional transfers. The method is quite similar to the way in which numeric comparisons are performed, using an IF. . . THEN. . . statement as before. The syntax form is essentially the same:

[Line number] IF numeric expression THEN line number

The only difference is the way the *numeric expression* is constructed, taking the form

character-string relational-operator character-string

where the *relational-operator*, as before, can be any of the six relational operators and *character-string* can be either a string constant or a string variable.

Some examples of string IF. . . THEN. . . statements:

```
100 IF A$=B$ THEN 500
110 IF A$>F$ THEN 600
120 IF B$="END" THEN 300
```

As mentioned earlier, characters are represented in the Graphic System using the ASCII code. In this scheme, each character in the set of 128 characters has a unique decimal equivalent within the range of 0 through 127. Thus, "A" is 65, "6" is 54, "!" is 33, etc. The fact that each character is represented numerically forms the basis for making string comparisons. You can say that "A" is less than "B" because "A", in ASCII decimal equivalence, is 65 while "B" is 66.

Character strings are evaluated on a character-by-character basis from left to right. The first character inequality discovered in the two strings determines the relationship. Consider the following two character strings:

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
①	P	A	R	E	N	T	H	E	S	I	S
②	P	A	R	E	N	T	H	E	S	E	S

In comparing these two strings, the BASIC interpreter determines that string ① is greater than string ② . This relationship is determined when the 10th pair of characters (the first inequality) is examined. The first 9 pairs are equivalent, but in the 10th pair the "I" has a numeric representation greater than the "E". No further comparisons are performed once the first inequality is located.

The SET CASE/SET NOCASE statements (mentioned in the DIRECTIVES section) affect string comparisons. When the default condition of SET CASE is in effect, lower case letters are temporarily converted to their upper case equivalents when string comparisons are made. When SET NO CASE is in effect, lower case and upper case letters are not equivalent in string comparisons. A tabulation of the ASCII character set is in the Appendix.

Concatenation

Concatenation is the process of linking (or joining) character strings together. If you concatenate a "T" and an "H" and an "E", you get "THE". Concatenation is accomplished with the concatenation operator, which is the ampersand symbol (&). You concatenate string constants as shown below:

```
100 A$="CHARACTER"&"STRING"
```

which yields "CHARACTERSTRING". Or, the operation can involve string constants and string variables, as in

```
100 A$=B$&"ES"
110 A$="ANTI"&B$
```

Also, concatenation can involve string variables like

```
100 A$=B$&C$
```

Concatenation takes the form

```
[Line number] target string = character string & character string
```

where the character string can be either a string constant or a string variable, and the target string must have a physical size of sufficient length to contain the result.

FUNCTIONS

Substrings

A substring is a segment of a character string. Thus, "AMPLE" is a substring of "EXAMPLE" and "HISTOWN" is a substring of "JOHN DOE, HISTOWN, OHIO". BASIC gives you the ability to identify, isolate, and compare segments of strings with an eye towards character string manipulation. This means that you can process mixed data consisting of numeric and non-numeric data. The number of words in a string can be determined by looking for a blank between words. You can identify sentences by looking for periods at the end.

The LEN Function

When a string contains an unknown number of characters, as might be the case when using the default string dimension of 72 characters, you can determine the current length (or number of characters) by employing the built-in *length* function. This is written and used in a manner which is essentially the same as the mathematical functions:

$$\text{LEN}\left\{\begin{array}{l}\text{string constant}\\ \text{string variable}\end{array}\right\}$$

The LEN function returns an integer indicating the number of characters, or working size, of the specified *string variable*. An example:

```
100 LET A$="ALPHABET"
110 LET X=LEN(A$)
```

In this case, X is assigned the number 8. You can use LEN as you would any other function. Continuing with the above example:

```
120 DIM B$(2*LEN(A$))
```

This statement dimensions a string variable (B$) with twice the physical size of A$. You can use LEN in IF. . . THEN. . . statements:

```
130 LET B$=A$&A$
140 IF LEN(B$)>72 THEN 160
150 GO TO 170
160 PRINT "B$ CONTAINS ";LEN(B$);" CHARACTERS"
170 END
```

Recalling that the Graphic System Display is 72 characters wide, line 140 checks to see if B$ is longer than the number of characters the screen can contain. Statement 130 concatenates A$ with itself.

The SEG Function

You can extract a substring from an existing string with the three-parameter SEG (for segment) function. This has the form:

```
[Line number] [LET] string variable = SEG ( { string constant } , numeric expression ,
                                            { string variable }
    numeric expression )
```

where the first numeric expression specifies the starting position of the substring within the string, and the second numeric expression specifies the number of characters contained in the substring. A short example:

```
100 READ A$
110 B$=SEG(A$,7,5)
120 DATA "HENRY JONES"
```

In this case, B$ is assigned the substring "JONES", which is a 5 character substring starting at the 7th position of A$ (counting the blank character). A further example:

```
100 A$="SUBSTRING"
110 FOR I=1 TO LEN(A$)
120 B$=SEG(A$,I,1)
130 C$=SEG(A$,1,I)
140 PRINT B$,C$
150 NEXT I
160 END
```

The output from the above example:

```
S                 S
U                 SU
B                 SUB
S                 SUBS
T                 SUBST
R                 SUBSTR
I                 SUBSTRI
N                 SUBSTRIN
G                 SUBSTRING
```

The POS Function

An additional character handling function allows you to search for the occurrence of a particular substring within a specified string. You can look for a letter, a word, a phrase, etc. The function that facilitates this capability is the POS (for position) function with the following form:

$$\text{POS} \; (\left\{ \begin{matrix} \text{string constant} \\ \text{string variable} \end{matrix} \right\} , \left\{ \begin{matrix} \text{string constant} \\ \text{string variable} \end{matrix} \right\} , \text{ numeric expression })$$

where the first string in the parentheses following POS is the string to be searched. The second string is the substring to be found. The numeric expression specifies a character position which is the starting location for the search. The POS function returns an integer which is the position of the first occurrence of the substring to be found within the string that is searched.

To illustrate the POS function, assume that you have previously dimensioned a string (A$) to a length of 26, and it contains the alphabet in the usual sequence:

Location	1	2	3	4	5	6	7	8	9	10	11	12	13
Contents (A$)	A	B	C	D	E	F	G	H	I	J	K	L	M

Location	14	15	16	17	18	19	20	21	22	23	24	25	26
Contents (A$)	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

The following statements return the indicated results:

Statement	Contents of P
100 P = POS(A$, "XYZ",1)	24
110 P = POS(A$, "H",1)	8
120 P = POS(A$,"HIJK",1)	8
130 P = POS(A$,"HIJK",12)	0
140 P = POS(A$,"BX",1)	0

The following program segment is an additional example of the POS and SEG functions, taking the name "JOHN DOE" and switching it around to "DOE, JOHN". The technique depends on a blank character separating the first and last names.

```
100 LET A$="JOHN DOE"
110 LET L=LEN(A$)
120 LET P=POS(A$," ",1)
```

L now contains the length of A$, and P contains a number indicating the position of the blank character separating the first and last name.

```
130 LET B$=SEG(A$,P+1,L-P)
```

"DOE" has now been isolated and placed in B$.

```
140 LET C$=SEG(A$,1,P-1)
```

"JOHN" has also been isolated at this point and placed in C$.

```
150 LET A$=B$&","
160 LET A$=A$&C$
```

A$ now contains the string "DOE,JOHN" and the switch is complete.

The short program below is more involved: it inputs a sentence, and then prints out a frequency table listing each letter of the alphabet that appears in the sentence, plus the number of times it is used.

```
100 DIM A$(26)
110 LET A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
120 PRINT "ENTER SENTENCE: ";
130 INPUT B$
140 L=LEN(B$)
150 REMARK--SEARCH FOR FIRST OCCURENCE OF EACH LETTER OF ALPHABET
160 FOR I=1 TO 26
170 S$=SEG(A$,I,1)
180 F=0
190 P=POS(B$,S$,1)
200 IF P=0 THEN 340
210 P1=P
220 F=F+1
230 REMARK--SEARCH FOR SUBSEQUENT OCCURENCES
240 P=POS(B$,S$,P1+1)
250 IF P=0 THEN 290
260 P1=P
270 F=F+1
280 GO TO 240
290 PRINT S$;
300 FOR J=1 TO F
310 PRINT "*";
320 NEXT J
330 PRINT
340 NEXT I
350 END
```

An example of the output:

```
ENTER SENTENCE: THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG'S BACK
A##
B##
C##
D##
E####
F#
G#
H##
I#
J#
K##
L#
M#
N#
O####
P#
Q#
R##
S#
T##
U##
V#
W#
X#
Y#
Z#
```

The REP Function

The SEG (for segment) function can be used to extract a substring from a string. An additional function allows you to do what is essentially the converse of the SEG function. Instead of extracting a substring from a string, you can replace a substring with another substring. This capability is given by the REP (for replace) function:

[Line number] [LET] string variable = REP ($\left\{\begin{array}{l}\text{string constant}\\ \text{string variable}\end{array}\right\}$, numeric expression ,
numeric expression)

The string to the right of REP is the actual replacement string that will alter the string variable to the left of REP. The first numeric expression specifies the starting position in the target string that will receive the replacement string. The second numeric expression specifies the number of characters in the target string that will be replaced. For example, consider the following:

```
100 Z$="ABCGHI"
110 A$="DEF"
120 Z$=REP(A$,4,0)
130 PRINT Z$
140 END
```

Following execution of this program, Z$ will contain "ABCDEFGHI". Statement 120 says, in effect, "Z$ receives A$ starting at position 4. No characters in Z$ are to be deleted, as indicated by 0 in the third parameter location". Since nothing is to be deleted, all of the original characters in Z$ are retained, and A$ is inserted. Z$ originally looked like the following:

character position	(1)	(2)	(3)	(4)	(5)	(6)
	A	B	C	G	H	I

After execution of line 120, it looks like this:

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
A	B	C	D	E	F	G	H	I

Notice that the working size of Z$ has been increased. This means that you have to make sure that the target string is dimensioned properly to accommodate the character insertion. The example uses the default dimension of 72, so size is not a problem in this case; however, you can see that conditions could exist where the size of the target string could be a problem.

The next example using REP does cause an actual replacement of characters in the target string (A$).

```
100 A$="CONFECTION"
110 A$=REP("CO",4,2)
120 PRINT A$
130 END
```

The previous program changes A$ from "CONFECTION" to "CONCOCTION". This time, statement 110 says "A$ receives "CO" starting at position 4. The 2 existing characters in position 4 and 5 are to be replaced". Notice that the length of A$ is unchanged in this instance.

The Graphic System version of BASIC does not include the ability to subscript string variables (as in A$(1), B$(N), etc.). In applications where this is necessary, the REP function does provide an approach toward simulating this ability. For instance:

```
100 A$="0"
110 B$="1"
120 FOR I=1 TO 19 STEP 2
130 C$=REP(A$,I,0)
140 C$=REP(B$,I+1,0)
150 NEXT I
160 PRINT C$
170 END
```

C$ contains alternating 0's and 1's. The effect is the same as if you had been able to write:

```
A$ = "0"
B$ = "1"
FOR I = 1 TO 19 STEP 2
C$(I) = A$
C$(I+1) = B$
NEXT I
     etc.
```

CONVERSIONS

The VAL Function

The situation can arise where you have some data to process, but it is not in the correct form. For instance, you might be confronted with ASCII numbers, which are really characters, when you need their decimal counterparts instead. Obviously, you can't expect to get the square root of the letter "7" because it is a character like "X" and "?". What is needed is the means to convert between ASCII numbers (character strings) and their numeric equivalents, and vice versa. This capability is provided by a pair of conversion functions called VAL (for value) and STR (for string).

The VAL function takes the form

```
VAL { string variable }
    { string constant }
```

where *string variable* refers to an existing character string which contains elements of the following character set:

The letters 0 through 9
The monadic operators "+" and "–"
The decimal point
The exponent symbol "E"

For example:

```
100 LET A$="123.4"
110 LET B=VAL(A$)
```

The above two statements cause the numeric variable B to receive the value 123.4. This example is, perhaps, an over simplification, because it is equivalent to writing

```
100 LET B=123.4
```

However, you can get the idea behind the VAL function.

The STR Function

The STR function is the converse of the VAL function, and takes the form

```
[Line number] [LET] string variable = STR (numeric expression)
```

The following short example is similar to the one used with VAL:

```
100 LET A=123.4
110 LET B$=STR(A)
```

The result of the above two lines is that B$ now contains "123.4", and this is equivalent to saying

```
100 LET B$="123.4"
```

The important distinction to be made here is the difference between "123.4" and 123.4. The number 123.4 can be used in math operations; the character string "123.4" can't be used in math operations. Also, it's important to realize that the conversion between the two forms can be readily made.

The ASC Function

In addition to the pair of functions VAL and STR, there is another pair of conversion functions called ASC (for ASCII) and CHR (for character). VAL and STR, you will recall, pertain to conversions between numbers and character strings representing numbers. ASC and CHR, on the other hand, provide a way to convert readily between ASCII characters and their decimal equivalents.

ASC takes the form

$$\text{ASC}\quad\begin{Bmatrix}\text{string constant}\\ \text{string variable}\end{Bmatrix}$$

where the *string variable* must be a one-character string. The character can be any one of the 128 standard ASCII characters. (A–Z, 0–9, etc., as listed in the ASCII character set in the Appendix.) The letter "A", for example, has an ASCII decimal equivalent of 65, and this number can be obtained using the ASC function:

```
100 LET A$="A"
110 LET B=ASC(A$)
```

B now contains 65. Incidentally, if A$ contained "8", ASC(A$) would return a 56, which is the ASCII decimal equivalent for the character "8".

The CHR Function

The CHR function is the converse of the ASC function, with the form

[Line number] [LET] string variable = CHR (numeric expression)

The numeric expression must reduce to a number between 0 and 127, because there are no ASCII characters with decimal equivalents outside this range. A short example:

```
100 LET B=65
110 LET A$=CHR(B)
```

This equivalent to saying

```
100 LET A$="A"
```

Summary of the Character Conversion Functions

Functions:	VAL and STR	ASC and CHR
Purpose:	Conversion between numbers and character strings representing numbers.	Conversion between a one-character string and its ASCII decimal equivalent.
Appropriate character set:	Letters 0–9 Monadic operators + and – Decimal point Exponent symbol "E"	Any one of the standard 128 ASCII characters

SUMMARY

The character handling capability of the Graphic System allows you to process non-numeric data. Character strings have two predominent properties: each string has a length; and each character in a string has a position number. Characters are represented in the Graphic System as ASCII equivalents.

Strings are assigned to string variables with the assignment operator. You can allocate memory for string variables with DIM; the default size for string variables is 72 characters.

String input/output is accomplished with INPUT, READ/DATA, and PRINT, much like the process for numeric I/O. Strings in DATA statements require enclosing quotes, string input with INPUT does not require enclosing quotes.

Controling program execution can be accomplished through string comparisons using IF. . . THEN. . . statements. Strings are evaluated left to right on a character-by-character basis, with the first inequality determining the difference.

Strings can be concatenated with the "&" operator. The destination string must be large enough to contain the result.

Substrings are portions of larger strings. You can identify, isolate, and compare substrings and strings through use of the various character handling functions. LEN lets you determine the length of a string or substring. Substrings can be extracted through the use of SEG, and you can insert or replace characters in a string with REP. POS allows you to locate the position of a substring within a string by returning a number indicating the starting position of the substring.

There is a difference between the "letters" 0–9 and the "numbers" 0–9; VAL and STR let you convert freely between the two forms. You can also convert freely between ASCII characters and their decimal equivalents with ASC and CHR.

EXAMPLE PROGRAMS

TITLE: **Decimal to Binary Conversion**

DESCRIPTION: The program inputs an integer m, 0 < m < 99999, and outputs the binary equivalent. Non-integer numbers, and numbers not in the range of 0—99999, are rejected. The binary equivalent is output as a character string.

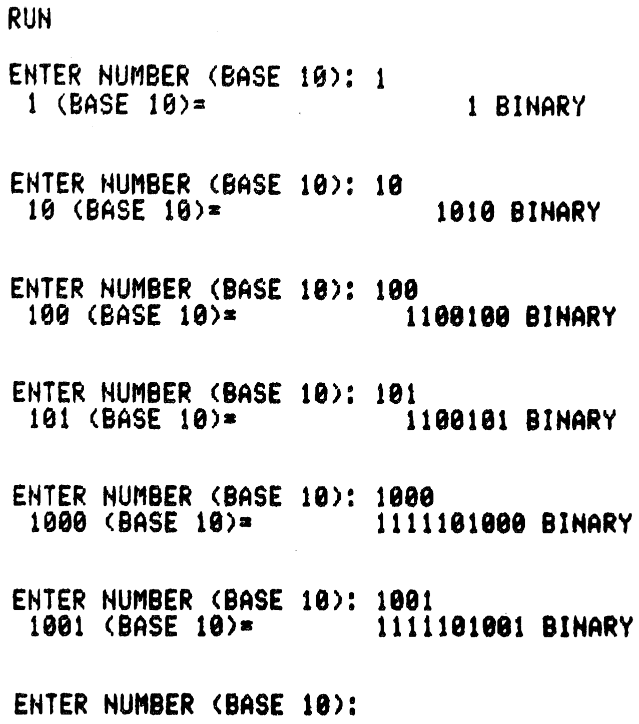
PROGRAM LISTING:

```
100 REMARK DECIMAL TO BINARY CONVERSION
110 PRINT
120 PRINT "ENTER NUMBER (BASE 10): ";
130 INPUT M
140 REMARK TEST TO VERIFY THAT M IS AN INTEGER, 0<M<99999
150 IF M<>INT(M) THEN 190
160 IF M<0 THEN 190
170 IF M>99999 THEN 190
180 GO TO 210
190 PRINT "NUMBER ENTERED INCORRECTLY"
200 GO TO 120
210 DIM B$(17),D$(2),T$(1)
220 REMARK INITIALIZE B$ TO 0 (BINARY)
230 B$="                0"
240 D$="01"
250 N=M
260 FOR B=17 TO 1 STEP -1
270 IF N=0 THEN 340
280 Q=INT(N/2)
290 D=N-Q*2+1
300 T$=SEG(D$,D,1)
310 B$=REP(T$,B,1)
320 N=Q
330 NEXT B
340 PRINT M;" (BASE 10)= ";B$;" BINARY"
350 PRINT
360 PRINT
370 GO TO 120
380 END
```

METHODOLOGY: The program manipulates character strings containing 1's and 0's to obtain binary representation of base 10 numbers.

OPERATING PROCEDURE: Type RUN, press RETURN, and the program responds with "ENTER NUMBER (BASE 10):". Enter the number you want converted to binary and press RETURN. If the number is not within the prescribed range (an integer, 0—99999) the message NUMBER ENTERED INCORRECTLY appears. Terminate the program by pressing BREAK BREAK.

OUTPUT SAMPLE:



```
RUN

ENTER NUMBER (BASE 10): 1
 1 (BASE 10)=                1 BINARY

ENTER NUMBER (BASE 10): 10
 10 (BASE 10)=            1010 BINARY

ENTER NUMBER (BASE 10): 100
 100 (BASE 10)=        1100100 BINARY

ENTER NUMBER (BASE 10): 101
 101 (BASE 10)=        1100101 BINARY

ENTER NUMBER (BASE 10): 1000
 1000 (BASE 10)=    1111101000 BINARY

ENTER NUMBER (BASE 10): 1001
 1001 (BASE 10)=    1111101001 BINARY

ENTER NUMBER (BASE 10):
```

TITLE: **Pig Latin**

DESCRIPTION: The program inputs a sentence, and outputs a Pig Latin version of the same sentence.

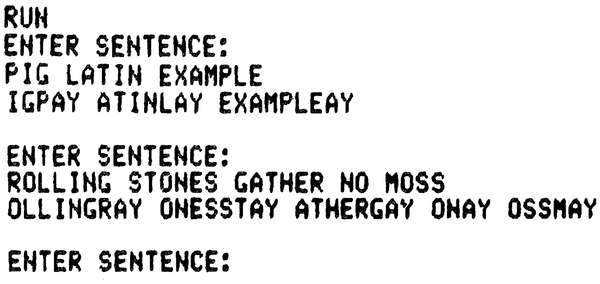
PROGRAM LISTING:

```
100 REMARK PIG LATIN PROGRAM
110 DIM P1(5)
120 PRINT "ENTER SENTENCE:"
130 INPUT A$
140 A$=A$&" "
150 I=0
160 L=LEN(A$)
170 REMARK  ISOLATE WORD
180 P=POS(A$," ",I+1)
190 B$=SEG(A$,I+1,P-I-1)
200 I=P
210 REMARK  LOOK FOR VOWELS, STORE LOCATIONS IN P1
220 P1(1)=POS(B$,"A",1)
230 P1(2)=POS(B$,"E",1)
240 P1(3)=POS(B$,"I",1)
250 P1(4)=POS(B$,"O",1)
260 P1(5)=POS(B$,"U",1)
270 REMARK GET RID OF ZEROS IN P1
280 FOR J=1 TO 5
290 IF P1(J)=0 THEN 310
300 GO TO 320
310 P1(J)=1000
320 NEXT J
330 P2=P1(1) MIN P1(2) MIN P1(3) MIN P1(4) MIN P1(5)
340 REMARK  ISOLATE PORTION OF WORD PAST VOWEL
350 C$=SEG(B$,P2,LEN(B$)-P2+1)
360 REMARK  ISOLATE PORTION OF WORD BEFORE VOWEL
370 D$=SEG(B$,1,P2-1)
380 REMARK  CONCATENATE BOTH PORTIONS AND ADD "AY "
390 E$=C$&D$
400 E$=E$&"AY "
410 PRINT E$;
420 IF I<L THEN 180
430 PRINT
440 PRINT
450 GO TO 120
460 END
```

METHODOLOGY: The sentence to be converted to Pig Latin is stored in A$. The program detects the end of a word by looking for the space that separates words. Line 140 concatenates a space on the end of the sentence so that the end of the last word may be located. Individual words are isolated with the POS and SEG statements (lines 180 and 190). The numeric variable "I" is used as a pointer to keep track of the location of the end of the word currently being converted. Once the end of a word is located, the first vowel is found and its position is stored. Using this position, the word is separated at the vowel, flipped around, and "AY" is concatenated on the end. Line 330 locates the position of the first vowel.

OPERATING PROCEDURE: Type RUN, press RETURN, and the program responds with ENTER SENTENCE. You then enter the sentence to be converted, press RETURN, and the Pig Latin version appears. The program then loops back and again responds with ENTER SENTENCE. Exit the program by pressing the BREAK key twice. Note: the sentence to be converted can be no longer than 72 characters.

OUTPUT SAMPLE:



```
RUN
ENTER SENTENCE:
PIG LATIN EXAMPLE
 IGPAY ATINLAY EXAMPLEAY

ENTER SENTENCE:
ROLLING STONES GATHER NO MOSS
 OLLINGRAY ONESSTAY ATHERGAY ONAY OSSMAY

ENTER SENTENCE:
```

Section 6

SUBROUTINES

WRITING SUBROUTINES

A subroutine is part of a larger ("main") program, and can be thought of as a subprogram. Subroutines are frequently used to perform some particular task that the main program repetitively requires. In this sense, subroutines can save you a lot of programming steps. Instead of having to place the same group of statements in several places in the program, you just situate the group at one place, make it into a subroutine, and direct the flow of program execution to the subroutine. Once flow reaches the subroutine, the particular task is performed, and program control returns to the appropriate place in the main program.

Conventional Subroutines

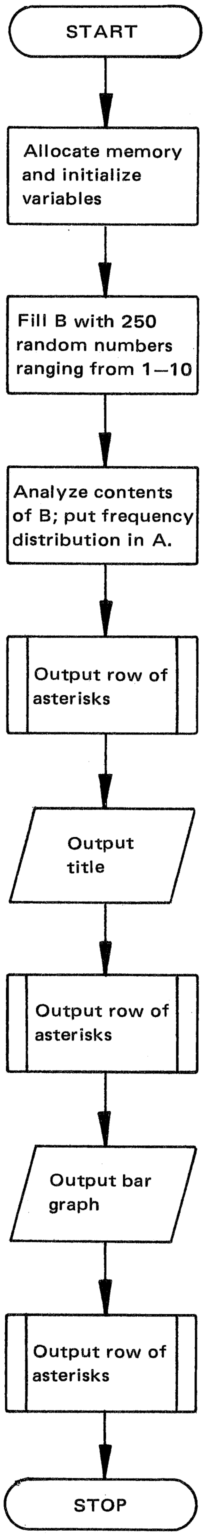
Two new statements are required for subroutines; their syntax forms are:

```
[Line number] GOSUB line number
[Line number] RETURN
```

The line number following GOSUB is the number of the first statement in the subroutine. GOSUB is conceptually quite similar to GOTO. The difference is that when the BASIC interpreter executes a GOSUB, it "remembers" the line number containing the GOSUB. Later, when the BASIC interpreter reaches the RETURN at the end of the subroutine, it transfers control back to the statement following the GOSUB.

The following program generates a simple bar graph displaying the distribution of 250 numbers ranging from 1 through 10. The numbers that are graphed are obtained through the use of the RND (for random) function and the INT (for integer) function, filling a 250 element array with random integers ranging from 1 through 10. This is done to simulate the otherwise tedious process of having to input all 250 numbers. Once the 250 numbers are generated and stored, the program goes on to determine how many times each number appears. After this is done, the bar graph is produced.

The important thing to look for in this program is the use of GOSUB. The subroutine itself isn't especially exotic — all it does is output a row of asterisks, and this is done only to dress up the output. A flowchart of the program appears below. Note the symbol that is used to represent the subroutine.


START
Allocate memory and initialize variables
Fill B with 250 random numbers ranging from 1—10
Analyze contents of B; put frequency distribution in A.
Output row of asterisks
Output title
Output row of asterisks
Output bar graph
Output row of asterisks
STOP

Array B (250 elements) will contain the data (all integers); array A (10 elements) will contain the number of occurrences of each integer in B. Initialize A to zero; initialize argument for RND function.

Use RND and INT functions in a loop to provide 250 integer values for B.

A (1) contains the number of 1's, A (2) contains the number of 2's, etc.

Call the subroutine that will generate the asterisks. (Note the symbol representing the subroutine.)

This prints "BAR GRAPH"

Call the subroutine again.

Use pound sign (#) symbols to form the graph.

Call the subroutine a third time.

The program listing appears below. The subroutine starts at line 360, and ends with the RETURN at line 400.

```
100 DIM A(10),B(250)
110 A=0
120 X=1
130 REM GENERATE,STORE, AND TALLY 250 RANDOM INTEGERS
140 FOR I=1 TO 250
150 B(I)=INT(10*RND(X))+1
160 A(B(I))=A(B(I))+1
170 NEXT I
180 GOSUB 360
190 PRINT
200 PRINT "BAR GRAPH"
210 GOSUB 360
220 PRINT
230 PRINT
240 REM GENERATE GRAPH
250 FOR I=1 TO 10
260 PRINT "NUMBER OF ";I;"'S",""; 
270 FOR J=1 TO A(I)
280 PRINT "#";
290 NEXT J
300 PRINT
310 NEXT I
320 PRINT
330 PRINT
340 GOSUB 360
350 GO TO 410
360 REM SUBROUTINE TO GENERATE A ROW OF 50 ASTERISKS
370 FOR I=1 TO 50
380 PRINT "*";
390 NEXT I
400 RETURN
410 END
```

The output produced by the above program looks like the following:

```
**************************************************
BAR GRAPH
**************************************************

NUMBER OF 1'S     ##########################
NUMBER OF 2'S     ########################
NUMBER OF 3'S     ###########################
NUMBER OF 4'S     ######################
NUMBER OF 5'S     ########################
NUMBER OF 6'S     ###########################
NUMBER OF 7'S     ####################
NUMBER OF 8'S     ############################
NUMBER OF 9'S     ##############################
NUMBER OF 10'S    ######################


**************************************************
```

The subroutine is "called" three times. If it had been called only once, it would not have saved any program steps, but since it is called three times, the program is shortened. The first call for the subroutine is at line 180. When the BASIC interpreter reaches this statement, it transfers control to the appropriate line number (which is 360 in this case). From statement 360, the statements which form the subroutine are executed in normal sequence until the RETURN statement is found. The RETURN causes the BASIC interpreter to transfer control back to the statement which follows the GOSUB, and program execution again proceeds in normal sequence. Notice the GOTO at line 350: this prevents the BASIC interpreter from reaching the subroutine except via the GOSUB statements. An accidental entry into the subroutine produces unpredictable results when the RETURN is executed.

The preceding program emphasizes the capability of subroutines to reduce the amount of programming that might otherwise be required. This, however, is not the only value of subroutines. They are also useful from the standpoint of allowing you to break up large programs into modules. This way, you can write each module separately as a subroutine, and then write the main program so that it calls the modules in when they are needed. You might have a main program that looks something like the following:

```
100 REMARK--INPUT AND STORE DATA
110 GOSUB 500
120 REMARK--PROCESS DATA
130 GOSUB 600
140 REMARK--OUTPUT DATA
150 GOSUB 700
160 END
```

The subroutines would be located beginning at the appropriate line numbers, each ending with a RETURN. The result is a clearly organized, modularized program.

For example, suppose you are writing a program that, among other things, needs to compute the factorial of a given number. A routine that does this is the following:

```
500 LET F=N
510 FOR M=N-1 TO 1 STEP -1
520 LET F=F*M
530 NEXT M
540 RETURN
```

Now, if your program needs the factorial of, say, 5 just assign the 5 to the variable N and call the subroutine:

```
100 N=5
110 GOSUB 500
120 PRINT F
```

Similarly, if the program needs the factorial of Y:

```
200 N=Y
210 GOSUB 500
220 PRINT F
```

Computed GOSUB

Earlier it was mentioned that GOSUB is conceptually similar to GOTO except for the associated RETURN statement. The GOTO statement can be written in a GOTO. . . OF. . . form. The similarity between GOTO and GOSUB continues in this vein. There is also a GOSUB. . . OF. . . form:

[Line number] GOSUB numeric expression OF line number [, line number] . . .

The numeric expression is rounded to the nearest integer n. That is, 3.7 becomes 4. If n is greater than the number of lines represented in the line number list, the GOSUB instruction is ignored. Also, if n is less than 1, the GOSUB instruction is ignored. The statements in the line number list must be separated by commas. If these conditions are satisified, the GOSUB transfers control to the nth line number in the line number list. Examples:

BASIC Statements	Results
500 GOSUB 3 of 1000, 2000,3000	Control is transferred to statement 3000
100 READ A,B,C 110 DATA 75,2,150,97124 120 GOSUB B of 500,600,700,800	Control is transferred to statement 600
500 READ X,X1 600 DATA 2.4,1.3 700 GOSUB X+X1 of 1000,2000, 3000,4000	X + X1 is rounded to 4, and control is transferred to statement 4000
1000 GOSUB 5 of 5000,6000,6000	The GOSUB is ignored because there are only 3 line numbers.
650 GOSUB –2 of 1000,2000	The GOSUB is ignored because the numeric expression is a negative number.

One further property of subroutines is that they can be nested. That is, one subroutine can call another subroutine. This necessitates a fair amount of care in constructing the routines. You must ensure proper placement of the RETURN statements, and safeguard against accidental entry into the subroutines. Remember that each subroutine requires an associated RETURN statement so the BASIC interpreter knows which line number it is supposed to return to.

Electronics people are familiar with the relationship between voltage, current, and resistance in a circuit. This relationship (Ohm's Law) can be stated algebraically as

$$E = IR$$

where E is the voltage, I is the current, and R is the resistance. With everything else held constant, if the current is increased, the voltage increases proportionally; similarly, if the resistance is increased, the voltage increases. The following program, based upon Ohm's Law, solves for E, I, or R when supplied with the other two ingredients. The program is included here becuase it calls subroutines which call other subroutines. Also, the program is "interactive" — that is, when executed, it asks you which parameter you want to solve for, and then asks you to enter values for the two missing parameters. Control is transferred to the appropriate subroutine through a character string indexing technique. The program is below:

```
100 PAGE
110 DIM A$(1),B$(3),C$(2)
120 B$="EIR"
130 PRINT
140 PRINT "ENTER PARAMETER TO BE FOUND (E,I,R)  ";
150 INPUT A$
160 N=POS(B$,A$,1)
170 IF N=0 THEN 280
180 REM GOTO APPROPRIATE SUBROUTINE
190 GOSUB N OF 300,360,420
200 C$="RT"
210 PRINT
220 PRINT "TO TERMINATE PROGRAM, ENTER T"
230 PRINT "TO RUN PROGRAM AGAIN, ENTER R"
240 INPUT A$
250 N=POS(C$,A$,1)
260 IF N=0 THEN 280
270 GO TO N OF 130,290
280 PRINT "INVALID ENTRY"
290 STOP
300 REM ROUTINE TO INPUT I&R, AND OUTPUT E
310 GOSUB 480
320 GOSUB 510
330 E=I*R
340 PRINT "VOLTAGE =";E
350 RETURN
360 REM ROUTINE TO INPUT E&R, AND OUTPUT I
370 GOSUB 540
380 GOSUB 510
390 I=E/R
400 PRINT "CURRENT =";I
410 RETURN
420 REM ROUTINE TO INPUT E&I, AND OUTPUT R
430 GOSUB 540
440 GOSUB 480
450 R=E/I
460 PRINT "RESISTANCE=";R
470 RETURN
480 PRINT "ENTER I (IN AMPS)  ";
490 INPUT I
500 RETURN
```

```
510 PRINT "ENTER R (IN OHMS)  ";
520 INPUT R
530 RETURN
540 PRINT "ENTER E (IN VOLTS)  ";
550 INPUT E
560 RETURN
570 END
```

The following is a sample of the output produced by the above program:

```
ENTER PARAMETER TO BE FOUND (E,I,R)  E
ENTER I (IN AMPS)  .005
ENTER R (IN OHMS)  20000
VOLTAGE =100

TO TERMINATE PROGRAM, ENTER T
TO RUN PROGRAM AGAIN, ENTER R
R

ENTER PARAMETER TO BE FOUND (E,I,R)  I
ENTER E (IN VOLTS)  200
ENTER R (IN OHMS)  5000
CURRENT =0.04

TO TERMINATE PROGRAM, ENTER T
TO RUN PROGRAM AGAIN, ENTER R
R

ENTER PARAMETER TO BE FOUND (E,I,R)  R
ENTER E (IN VOLTS)  500
ENTER I (IN AMPS)  .0025
RESISTANCE=200000

TO TERMINATE PROGRAM, ENTER T
TO RUN PROGRAM AGAIN, ENTER R
T

STOP IN LINE 290 PRIOR TO LINE 300
```

The program uses one of three "main" subroutines, depending on whether you want to solve for E, I, or R. Control is passed to the appropriate subroutine by an indexing technique which works as follows:

- B$ is assigned the characters "E I R" (line 120)

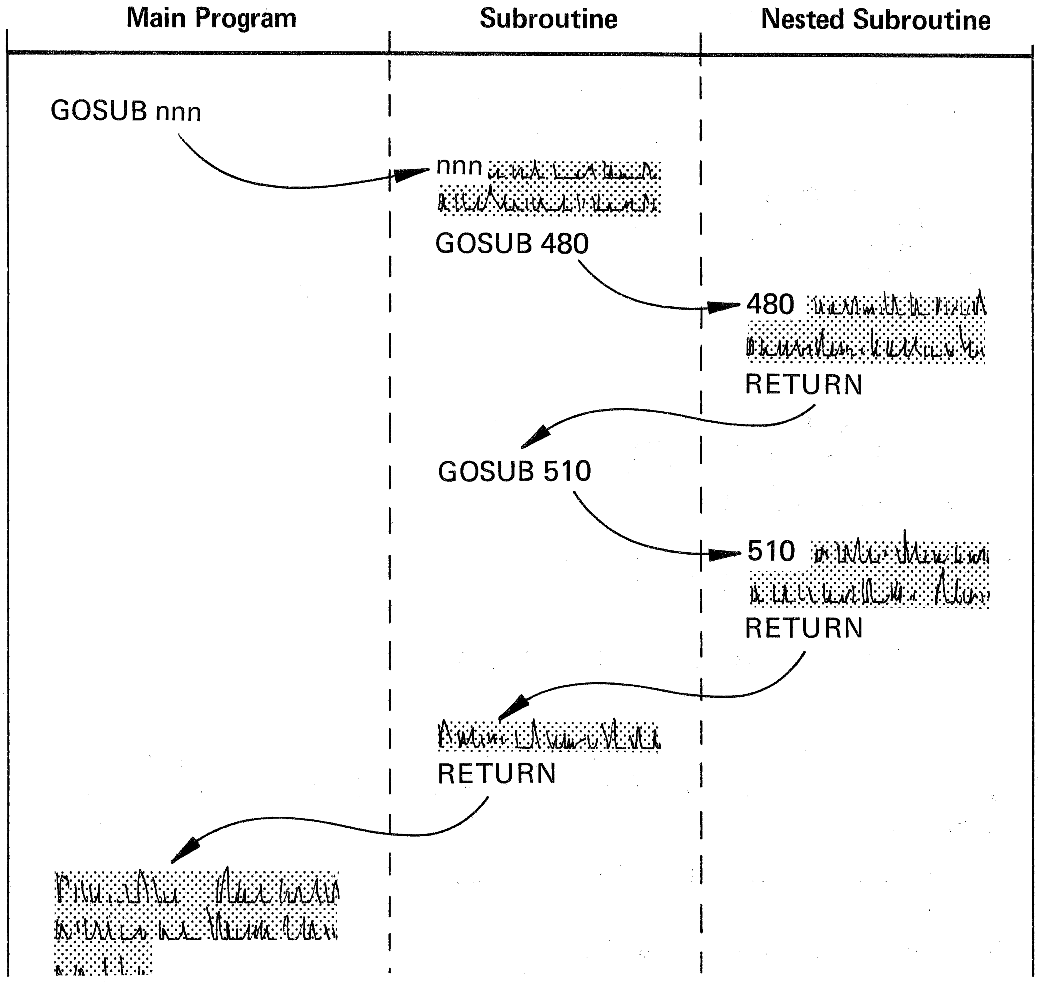
- A$ receives an E, I, or R in line 150.

- N becomes the *index* (a number used to indicate the position) of A$ within B$. This index value is 1, 2, or 3. (N could also be 0 if you input something other than E, I, or R into A$.)

- Line 190 then transfers control to the Nth subroutine specified.

Notice, in line 170, that provision has been made to handle erroneous entries. If you press a wrong key when you are supposed to enter E, I, or R, then the program responds with an error message.

The three subroutines start at lines 300, 360, and 420. Each of these three subroutines call other ("nested") subroutines which input values for E, I, or R. (The purpose of this particular approach is to illustrate the nesting of subroutines, and the program modularity which results.) Notice that each subroutine ends with a RETURN which transfers control back to the statement following the GOSUB. This nesting can be conceptalized as follows:



Once program control returns from the subroutines to the main program at line 200, it encounters a short routine which asks you if you want to run the program again or terminate execution. An indexing technique similar to the earlier one then transfers control to the appropriate statement in the program.

USER DEFINABLE KEYS

General

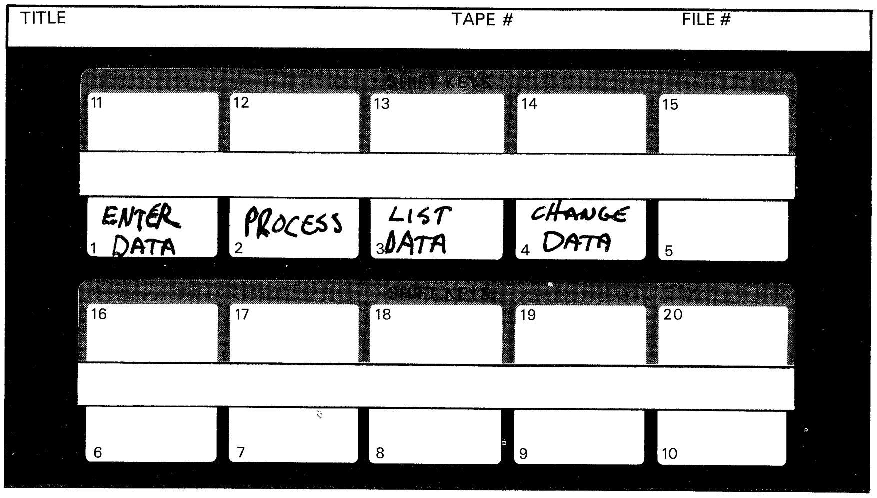
There are 10 actual User Definable Keys on the Graphic System keyboard. The use of the SHIFT key effectively doubles the number of keys to 20.

Obviously, these keys can not accomplish anything unless there is a program segment in memory that corresponds to the intended function of each key. What is not so obvious is that the program segment which corresponds to a given key must begin with a specific line number. User Definable Key (UDK) number 1, for instance, depends on a program statement being located beginning at line number 4. UDK number 2 requires a program statement beginning at line number 8, UDK number 3 requires a program statement at line 12, UDK number n required a program statement at line (n * 4). Pressing one of these keys is roughly equivalent to "GOSUB (Keynumber * 4)". Notice that only 4 lines of programming are allocated per key: this does not leave much room. The solution, of course is to place a GOTO or GOSUB statement at line (n * 4), where n is the UDK number. This way, the flow of execution is directed to an appropriate routine located elsewhere in memory.

An Illustrative Exercise

Let's suppose that you want to write a program that can (1) input data, (2) output a statistical summary of the data, (3) list out the data for examination, and (4) provide a convenient way to change individual data items if desired. Each of these 4 actions is to be initiated by pressing a User Definable Key.

With the major objectives of the program defined (above), the next step is to define, through use of a keyboard overlay, which keys do what. The following is suitable:



The first four User Definable Keys are used as indicated by the overlay. From this beginning it is clear that the program must have control statements (like GOTO or GOSUB) at lines 4, 8, 12, and 16 to direct the flow of execution to the appropriate subroutines. The "appropriate subroutines" do not yet exist, of course, but let's assume that the routine for "Enter Data" starts at line 1000, and the other three routines have starting line numbers of 2000, 3000, and 4000. The following statements can now be written:

```
4 GO TO 1000
8 GO TO 2000
12 GO TO 3000
16 GO TO 4000
```

Now, when UDK number 1 is pressed, control is transferred to line 1000. UDK number 2 results in control being passed to line 2000, and so on.

Writing a subroutine for the "Enter Data" key can now be started. First a specification of the action that is to occur when the key is pressed:

- The program asks how many data items there are. This information is used in dimensioning the array that is to receive the data.
- After allocating space for the array, the program signals you to enter the data.
- When the data array has been filled, the program will issue an appropriate message.

The following lines satisfy the above specifications.

```
1000 REM ***********ROUTINE TO ENTER DATA***********
1010 PRINT
1020 PRINT "ENTER NUMBER OF DATA ITEMS: ";
1030 INPUT N
1040 DELETE A
1050 DIM A(N)
1060 PRINT "ENTER DATA"
1070 FOR I=1 TO N
1080 PRINT "          ";"(";I;")"," ";
1090 INPUT A(I)
1100 NEXT I
1110 PRINT "DATA ENTRY TERMINATED"
1120 RETURN
```

The routine asks how many data items there are (line 1020); this figure is used to dimension array A in line 1050. You may be wondering about the DELETE statement in line 1040. Suppose that you run the program once and specify 10 data items, and then re-run the program and specify 20 data items. Recall that redimensioning of arrays can only make the array smaller, not larger. Since, in this routine, the number of data items affects the size of the array, the re-run would attempt to redimension A from 10 elements to 20 elements and result in an error; the DELETE circumvents this possibility. The subroutine signals the keyboard operator to begin entering data in line 1060. A FOR. . , NEXT. . . loop is used to input the data. The PRINT in line 1080 provides indented data item reference numbers; i.e., this will look like

(1) data item

(2) data item

(3) data item

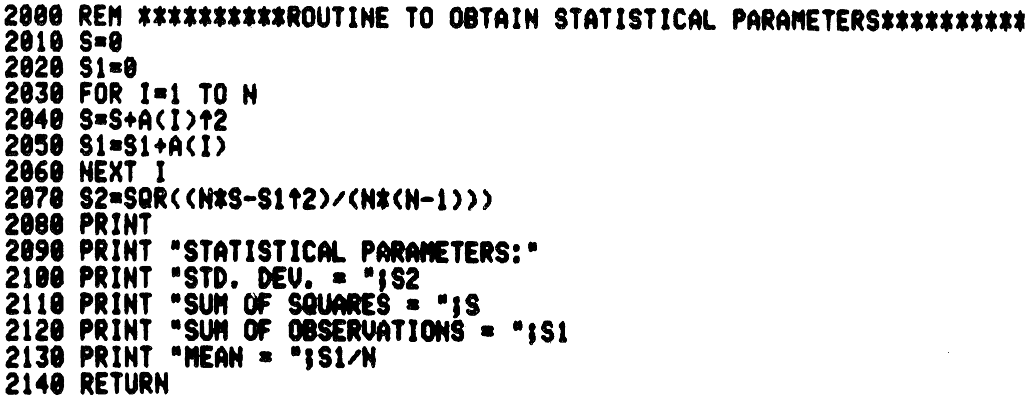
etc

When the indicated number of data items have been entered, an appropriate message is issued (line 1110). Notice, in line 1120, the use of a terminating RETURN. Used in this manner, the RETURN simply returns control of the Graphic System from the program to the keyboard.

Now, having writtén the portion of the program that supports the "Enter Data" key, let's do the same thing for the "Process" key. First, to specify what this routine does:

- Determine the sum and sum of the squares of data.
- From the above, obtain the standard deviation and the mean of the data.
- Output the standard deviation, sum of the squares, sum of the observations, and the mean.

Recalling that the data is stored in array A, the program segment to implement these specifications is relatively straightforward:



```
2000 REM **********ROUTINE TO OBTAIN STATISTICAL PARAMETERS**********
2010 S=0
2020 S1=0
2030 FOR I=1 TO N
2040 S=S+A(I)↑2
2050 S1=S1+A(I)
2060 NEXT I
2070 S2=SQR((N*S-S1↑2)/(N*(N-1)))
2080 PRINT
2090 PRINT "STATISTICAL PARAMETERS:"
2100 PRINT "STD. DEV. = ";S2
2110 PRINT "SUM OF SQUARES = ";S
2120 PRINT "SUM OF OBSERVATIONS = ";S1
2130 PRINT "MEAN = ";S1/N
2140 RETURN
```

Lines 2010 and 2020 initialize some variables that are used as accumulators. The loop of lines 2030 – 2060 obtains the sum of the data and the sum of the squares of the data. Statement 2070 supplies the standard deviation. The output of this routine is given by statements 2080–2130. The routine terminates with a RETURN, restoring control to the Graphic System keyboard.

Specifications for the third (List Data)key:

- Furnish a heading ("DATA LISTING:") to identify the routine.
- Output the data, giving data item reference numbers.

The code:

```
3000 REM **********ROUTINE TO LIST DATA**********
3010 PRINT
3020 PRINT "DATA LISTING:"
3030 FOR I=1 TO N
3040 PRINT "          ";"(";I;")",A(I)
3050 NEXT I
3060 RETURN
```

Again, the program is quite straightforward. The PRINT statement within the loop indents the output by first outputting a series of spaces. This routine also terminates with a RETURN.

This leaves only one more key requiring a supportive program segment. As before, let's first specify what the routine is to accomplish:

- It needs to ask which data item requires changing.
- A test should be done to verify that the operator does not erroneously enter an invalid (i.e., non-existent) data item reference number. This prevents potential error conditions from occurring.
- The routine should then input the new value for the specified data item.

The routine:

```
4000 REM **********ROUTINE TO CHANGE DATA**********
4010 PRINT
4020 PRINT "WHICH DATA ITEM REQUIRES CHANGING? ";
4030 INPUT M
4040 IF M<1 OR M>N THEN 4060
4050 GO TO 4080
4060 PRINT "INVALID ITEM REFERENCE"
4070 GO TO 4020
4080 PRINT "ENTER NEW VALUE FOR DATA ITEM ";M
4090 INPUT A(M)
4100 RETURN
```

Line 4020 asks for the number of the data item that is to be changed; the number is stored in variable m. Statement 4040 determines the validity of the entered number. If the number is less than 1, it's obviously not suitable as an array subscript because it is not a positive integer. Also, if it's greater than the number of data items originally input (N), it's again not valid. Line 4090 inputs the new data value, and the RETURN at line 4100 terminates the routine and completes the program.

Now, lets run the program. With the overlay in place press the key marked "Enter Data". The first thing that happens is that the program asks you to enter the number of data items you are going to be working with. The example output that follows uses 10 data items:

```
ENTER NUMBER OF DATA ITEMS: 10
ENTER DATA
            (1)       12
            (2)       23
            (3)       34
            (4)       45
            (5)       56
            (6)       67
            (7)       7
            (8)       78
            (9)       89
            (10)      90
DATA ENTRY TERMINATED
```

Notice the message which appears after the last data item has been entered. The statistical parameters of the just-entered data can be obtained by pressing "Process":

```
STATISTICAL PARAMETERS:
STD. DEV. = 30.7912325184
SUM OF SQUARES = 33633
SUM OF OBSERVATIONS = 501
MEAN = 50.1
```

To examine the data, press "List Data":

```
DATA LISTING:
            (1)       12
            (2)       23
            (3)       34
            (4)       45
            (5)       56
            (6)       67
            (7)       7
            (8)       78
            (9)       89
            (10)      90
```

Individual data items can be changed through use of the "Change Data" key. Suppose, for instance, that data item (7) is supposed to be 17, not 7. This change can easily be entered by pressing "Change Data":

```
WHICH DATA ITEM REQUIRES CHANGING? 7
ENTER NEW VALUE FOR DATA ITEM 7
17
```

Pressing "List Data" now results in the following list:

```
DATA LISTING:
          (1)       12
          (2)       23
          (3)       34
          (4)       45
          (5)       56
          (6)       67
          (7)       17
          (8)       78
          (9)       89
          (10)      90
```

Incidentally, if the "Change Data" key had been pressed and an invalid data item (like 12) has been entered, the output would read like the following:

```
WHICH DATA ITEM REQUIRES CHANGING? 12
INVALID ITEM REFERENCE
WHICH DATA ITEM REQUIRES CHANGING?
```

Data item 12, of course, doesn't exist, and the 12th element in the data array doesn't exist either because the array was dimensioned earlier to contain 10 elements. The program has been designed to catch this particular error which avoids the error message normally issued by the BASIC interpreter.

SUMMARY

A subroutine is a part of a larger main program. Subroutines can reduce the length of a program that requires some operation to be performed repetitively, because you only need to include the operation at one place in the program. Also, subroutines provide for modular programming.

Program execution reaches a subroutine through the GOSUB statement. Program flow is returned to the appropriate statement in the main program with RETURN. Subroutines can be nested so that one subroutine "calls" another subroutine.

Computed GOSUB arrangements are made possible with the GOSUB. . . OF. . . statement. These subroutines are also exited with RETURN.

When you press a User Definable Key, the Graphic System interprets this as meaning "GOSUB (key-number *4)". Thus, key number 1 is equivalent to GOSUB 4; key number 10 is equivalent to GOSUB 40, etc.

EXAMPLE PROGRAMS

TITLE: Roman Numeral Equivalents

DESCRIPTION: This program inputs a Roman Numeral and outputs the decimal equivalent. Each character in the character string containing the Roman Numeral is converted to its ASCII decimal equivalent, and this forms the basis for analyzing the Roman Numeral.

PROGRAM LISTING:

```
100 REMARK ROMAN NUMERAL EQUIVALENTS
110 PRINT "ENTER ROMAN NUMERAL: ";
120 INPUT R$
130 L=LEN(R$)
140 DELETE S
150 DIM S(L)
160 REMARK CONVERT CHARACTERS TO ASCII EQUIVALENTS
170 FOR I=1 TO L
180 A$=SEG(R$,I,1)
190 S(I)=ASC(A$)
200 NEXT I
210 GOSUB 250
220 PRINT "DECIMAL EQUIVALENT= ";S1
230 PRINT
240 GO TO 640
250 REMARK SUBROUTINE TO EVALUATE ROMAN NUMERAL
260 S1=0
270 P=1001
280 FOR I=1 TO L
290 REMARK TEST FOR "I"
300 IF S(I)<>73 THEN 330
310 D=1
320 GO TO 580
330 REMARK TEST FOR "V"
340 IF S(I)<>86 THEN 370
350 D=5
360 GO TO 580
370 REMARK TEST FOR "X"
380 IF S(I)<>88 THEN 410
390 D=10
400 GO TO 580
410 REMARK TEST FOR "L"
420 IF S(I)<>76 THEN 450
430 D=50
440 GO TO 580
450 REMARK TEST FOR "C"
460 IF S(I)<>67 THEN 490
470 D=100
480 GO TO 580
490 REMARK TEST FOR "D"
500 IF S(I)<>68 THEN 530
510 D=500
520 GO TO 580
530 REMARK TEST FOR "M"
540 IF S(I)=77 THEN 570
550 PRINT "INVALID SYMBOL"
```

```
560 END
570 D=1000
580 S1=S1+D
590 IF D<=P THEN 610
600 S1=S1-2*P
610 P=D
620 NEXT I
630 RETURN
640 END
```

METHODOLOGY: The characters forming the Roman Numerals are converted to their ASCII decimal equivalents. From there, a subroutine evaluates the Roman Numeral by testing for the presence of the various ASCII numbers. The variable S1 is used to accumulate the sub-totals as the Roman Numeral is evaluated. Line 590 checks to see if a symbol representing a "smaller" number occurs to the left of a symbol representing a "larger" number, as in IX or XC.

OPERATING PROCEDURE: Type RUN and press RETURN. Enter the Roman Numeral and press RETURN. Non-valid Roman Numeral symbols result in the message "INVALID SYMBOL".

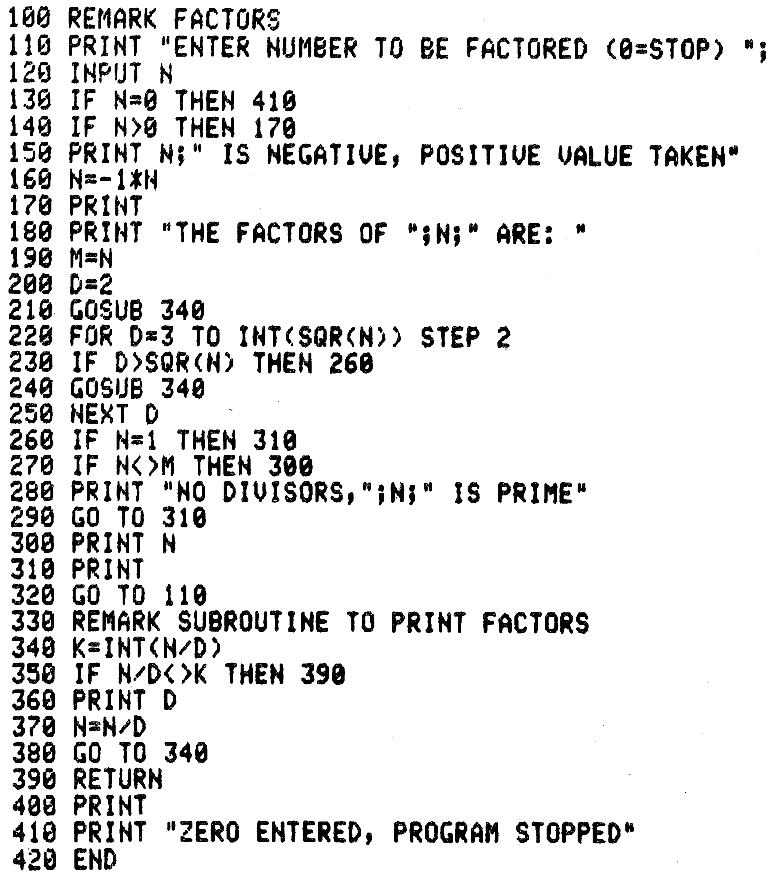
OUTPUT SAMPLE:

```
RUN
ENTER ROMAN NUMERAL: MDCCLXXVI
DECIMAL EQUIVALENT= 1776

RUN
ENTER ROMAN NUMERAL: MCMLXXVI
DECIMAL EQUIVALENT= 1976
```

TITLE: **Factoring Program**

DESCRIPTION: The program inputs an integer and outputs its factors.

PROGRAM LISTING:



```
100 REMARK FACTORS
110 PRINT "ENTER NUMBER TO BE FACTORED (0=STOP) ";
120 INPUT N
130 IF N=0 THEN 410
140 IF N>0 THEN 170
150 PRINT N;" IS NEGATIVE, POSITIVE VALUE TAKEN"
160 N=-1*N
170 PRINT
180 PRINT "THE FACTORS OF ";N;" ARE: "
190 M=N
200 D=2
210 GOSUB 340
220 FOR D=3 TO INT(SQR(N)) STEP 2
230 IF D>SQR(N) THEN 260
240 GOSUB 340
250 NEXT D
260 IF N=1 THEN 310
270 IF N<>M THEN 300
280 PRINT "NO DIVISORS,";N;" IS PRIME"
290 GO TO 310
300 PRINT N
310 PRINT
320 GO TO 110
330 REMARK SUBROUTINE TO PRINT FACTORS
340 K=INT(N/D)
350 IF N/D<>K THEN 390
360 PRINT D
370 N=N/D
380 GO TO 340
390 RETURN
400 PRINT
410 PRINT "ZERO ENTERED, PROGRAM STOPPED"
420 END
```

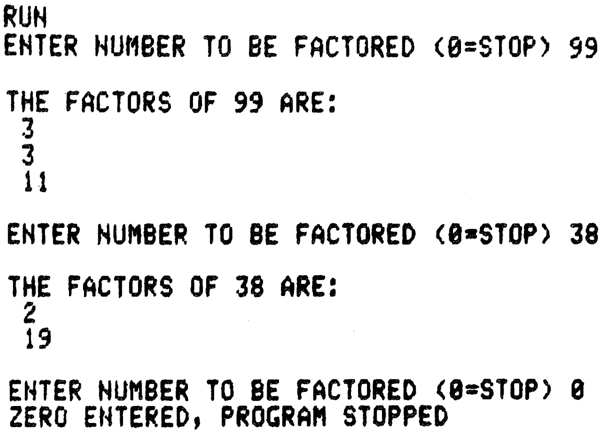
METHODOLOGY: The numeric variable N receives the value to be factored, and this value is then stored in M so that N can be manipulated. The first attempt to find a factor uses the value of 2 in the numeric variable D. The subroutine starting at line 340 then determines if D is a factor of N by testing to see if N/D is an integer. If N/D is an integer, then D is a factor of N, and D is output as a factor. Then, N receives the value of N/D, and the subroutine is repeated. When N/D results in a non-integer, control returns to the main program at line 220.

Once control reaches line 220, it encounters a FOR/NEXT loop which increments D from 3 to $\sqrt{N}$. Each value of D is examined by the subroutine to see if it is a factor of N, using the process outlined above. Recall that the body of a FOR/NEXT loop is always executed at least once. Line 230 is included to prevent control from reaching the subroutine in cases where $D > \sqrt{N}$; that is, in cases where 3 is already larger than $\sqrt{N}$ and it is not desirable to have the entire body of the loop executed.

Control is returned to the beginning of the program in one of two ways. Either N finally becomes equal to 1 (line 260) and control is routed back to line 110, or N is found to have no factors and is still equal to M (line 270). If N still equals M, an appropriate message results (line 280) and control is then routed back to statement 110.

OPERATING PROCEDURE: Type RUN and press RETURN. The program responds with "ENTER NUMBER TO BE FACTORED (0 = STOP)". Enter the desired number, press RETURN, and either the factors of the number are output or the message "NO DIVISORS, n IS PRIME" appears. The program then repeats with "ENTER NUMBER TO BE FACTORED (0 = STOP)". Repeat the above process, or enter 0 to terminate execution.

OUTPUT SAMPLE:



```
RUN
ENTER NUMBER TO BE FACTORED (0=STOP) 99

THE FACTORS OF 99 ARE:
 3
 3
 11

ENTER NUMBER TO BE FACTORED (0=STOP) 38

THE FACTORS OF 38 ARE:
 2
 19

ENTER NUMBER TO BE FACTORED (0=STOP) 0
ZERO ENTERED, PROGRAM STOPPED
```

Section 7

EXTENDED I/O

OUTPUT FORMATTING

Background Information

Up to this point, your output has been centered around the PRINT statement. You have been able to modify the appearance and format of the output by using a comma to generate a tab, a semicolon to suppress the tab, "bare" PRINT statements to place blank lines on the screen, and so on. The amount of control you have been able to exert over output thus far has been minimal.

The Graphic System has the facilities to provide extensive control over the output format; this, in turn, provides maximum output flexibility. You can exercise detailed control over the organization and appearance of the output. On the surface, the methods for implementing this control appear complex. However, once understood, the construction of output statements becomes easily accomplished.

Formatting the output depends on two things: the PRINT USING statement, which is an extended version of PRINT, and *format strings*. A format string can be included as part of the PRINT USING statement, or it can be included with another statement called the IMAGE statement. The *format string*, whether specified via PRINT USING or IMAGE, accomplishes the same thing: it determines the format of the output. Characters in the format string have special meaning. The essence of the output format rests, therefore, in the correct use of the various characters in the format string.

Before examining the ingredients that make up a format string, it is appropriate to become familiar with the two statements (PRINT USING and IMAGE) mentioned earlier. PRINT USING takes the following form:

```
                            { line number   }   { variable-list      }
[Line number] PRINT USING   {               } : {                    }
                            { format string }   { numeric expression }
```

The second line number refers to the line number of an IMAGE statement. The *format string* can be assigned to a string variable like A$, or it can be enclosed in quotation marks and placed directly in the PRINT USING statement. The remaining terms, *variable-list* and *numeric expression*, have the same meaning as in a conventional PRINT statement.

Examples:

```
100 PRINT USING 200:A,B,C
150 PRINT USING 250:3*(X↑2),T/N
```

The IMAGE statement has the following form:

```
[Line number] IMAGE format string
```

An example of a format string is 10A,5X (this will be elaborated upon later), so that a complete IMAGE statement looks like this:

```
200 IMAGE 10A,5X
```

Notice that the format string with the IMAGE statement does not require enclosing quotes. A complete PRINT USING/IMAGE combination therefore looks like this . . .

```
100 PRINT USING 200:A$
200 IMAGE 10A,5X
```

When executing the PRINT USING statement, the BASIC interpreter first examines the format string in the IMAGE statement, and then outputs the variable A$ according to the "picture" provided by the format string.

It makes no difference in the appearance of the output whether you use PRINT USING followed by a format string or use PRINT USING followed by the statement number of the IMAGE statement. The advantage of having an IMAGE statement is that several PRINT USING statements can refer to a single IMAGE statement, which avoids the necessity to repeat format strings or to commit string variables to the task of storing format strings.

Elements in the format string called *field operators* determine the output format. Different field operators produce different effects. In addition, some field operators have *modifiers* associated with them. The various field operators and modifiers are discussed in the following paragraphs.

The Character String Field Operator — A

The 'A' field operator controls the field width of character string output. That is, if you want to output n characters, you include within your format string an expression of the form nA. For instance, to output 5 characters you specify 5A; to output 20 characters, you say 20A, etc. The "n" in "nA" a *modifier.*

Let's suppose you have a program that outputs six columns of information, and you want to place headings (character strings) at the top of each column. One way to approach this is to remember that the screen can contain 72 characters on one line. If the output can suitably be centered on the screen, the screen can be divided up into 6 fields of 12 characters each, one field for each heading and associated column. With this in mind, the headings can be placed with output statements like the following:

```
100 PRINT USING 110:"ITEM","COLOR","WEIGHT","QUANTITY","PRICE","TOTAL"
110 IMAGE 12A,12A,12A,12A,12A,12A
```

When the Graphic System executes the PRINT USING statement, it first looks at the format string. Each specification (each field operator) is extracted and examined. In this case, the first specification calls for a character string, so the machine takes the first variable in the PRINT USING variable list and prints it according to specification. It then takes the second specification and prints the second item in the PRINT USING variable list according to specification, and so on. There must be a field operator for each item in the variable list or you will be presented with an error message. Incidentally, the commas in line 110 are optional and do nothing other then lend visual organization to the statement. More on this later.

Now, let's scrutinize statements 100 and 110 (above) again. There are six character strings in line 100 that are to be printed on the screen. The lengths of these six strings vary from four to eight characters. The format string specified in the IMAGE statement of line 110 says that the machine is to output six character string groups, and each group is to contain 12 characters. How, you might ask, do you get away with specifying groups of 12 characters, when the character strings themselves have lengths ranging from 4—8? Simple. As you might have anticipated, the machine outputs spaces, or blanks, as pad characters to the right of the actual string characters. The output from lines 100 and 110 above looks like the following:

```
ITEM        COLOR       WEIGHT      QUANTITY    PRICE       TOTAL
```

Before, you were able to generate only four columns of information on the screen by using the comma to generate a tab. Now, you can generate as many columns as you can fit. There is one thing to be careful about, however. If you make a specification in a format string like 8A (output 8 characters) and the actual character string contains more than 8 characters, a condition known as a field overflow will prevail, resulting in an error message.

In addition to the "n" modifier (like 8A and 12A), there is an F modifier that can associate with the A field operator. A specification like FA will cause the machine to output only those characters actually in the corresponding string. In this respect, the following two statements are similar:

```
PRINT USING "FA" : A$
PRINT A$
```

Now, to quickly summarize the A field operator:

- The A field operator controls the field width of character string output.
- nA outputs n characters. If the string is shorter than n, then spaces are output to the right of the actual string characters until n characters have been output. If the string is longer than n characters, a field overflow error results.
- FA outputs only those characters actually in the string.

Repeat Field Operator — ( )

In the earlier discussion about the character string field operator, there was the following statement:

```
110 IMAGE 12A,12A,12A,12A,12A,12A
```

The statement was written that way because it was necessary to provide a format specification for each variable being output. However, that method could quickly result in long, cumbersome format strings. To avoid this kind of potential difficulty, there are two "repeat" field operators: a "begin repeat" which is a left parenthesis, and an "end repeat" which is a right parenthesis. These two field operators allow you to write

```
110 IMAGE 6(12A)
```

instead of the unwieldy

```
110 IMAGE 12A,12A,12A,12A,12A,12A
```

Note that there is an n modifier associated with the "begin repeat" field operator: a 6 in this case. The format specifications within the repeat operators will be repeated n times. As you would expect, if you say IMAGE 6(10A) or something similar, there had better be 6 items (of the correct type) in the PRINT USING variable list or an error message will result.

Parentheses may be nested to a maximum of 4 deep. "n" must be a positive integer between 1 and 255; the default is 1.

Digit Field Operator — D

The discussion about field operators thus far has covered the formatting of character strings and the means of repeating format specifications. In the paragraphs that follow, the manipulation of numeric output (excluding, for the moment, scientific notation) is addressed. There are several combinations of things that can be done to affect the format of numbers: things like determining the number of digits that will appear in the output, the inclusion of a dollar sign and numeric sign, and the inclusion of commas (as in 1,234,567) as separators.

The writing of format specifications for numeric output requires that you pay attention to the width of the field. Consider the following cases:

Number	Field Width
456	3
−456	4 (The sign counts as 1)
4.56	4 (The decimal also counts)
+4.56	5
1456	4
1,456	5 (The comma counts as 1)
$1,456.00	9 (The dollar sign takes one position)

Having been alerted to the field width aspect of formatting, let's examine the various possibilities that exist with D and the modifiers that make it useful.

The D field operator accepts a "n" modifier just like the A field operator does. The range of values that "n" can assume is 1 to 255. The specification "6D" will output a number with a 6 digit field. The number will be an integer because the specification includes no information about where to place the decimal point. Also, the number will be right justified, and any leftmost unused digits will be padded with spaces. Let's look at a brief example of the D field operator with a "n" modifier:

```
100 X=PI↑10
110 PRINT X
120 PRINT USING "6D":X
130 END
```

Line 100 assigns an arbitrary value to X. Line 110 says to output the value of X, whatever that might be. Line 120 also says to output X, but it tells the machine to *edit* the output so that only 6 characters, including padded spaces, will appear on the screen. The output looks like the following:

```
93648.0474761
93648
```

The formatted, or edited, output (above) contains five digits, rounded to the left of the decimal. The format string produces a 6 character field. As a result, the unused character position to the left has been padded with one space to fill the 6 character field. If it had happened that the variable being output contained a number that was greater than 6 digits, a field overflow error would occur. Also, if the number had been negative and greater than 5 digits, a field overflow would occur because the minus sign takes up one character position.

In addition to the "n" modifier you can use a "." modifier to control the position of the decimal point. Let's change the format string in the previous example to include the "." modifier. The example now looks like this:

```
100 X=PI↑10
110 PRINT X
120 PRINT USING "6D.":X
130 END
```

The output now looks like the following:

```
93648.0474761
93648.
```

Notice the presence of the trailing decimal point. Also, the significance of the output goes beyond the inclusion of the decimal point: the field width is now 7. That is because the format string calls for 6 positions to contain numeric characters (or padded spaces) and 1 position for the decimal point.

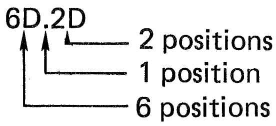
Let's modify the example again to cause the output to contain a decimal portion:

```
100 X=PI↑10
110 PRINT X
120 PRINT USING "6D.2D":X
130 END
```

The output this time looks like the following:

```
93648.0474761
93648.05
```

The field width in this case is 9:



The number has been rounded, in this instance, at the second place to the right of the decimal. Rounding always occurs at the nth place to the right of the decimal.

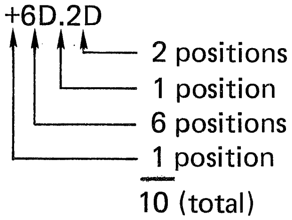
The "D" field operator has been shown so far in conjunction with the "n" modifier and the "." modifier. It can also be used with "–" and "+" modifier. Recall that something like "6D" allows for a 6 digit field in the case of positive integers, and room for 5 digits plus a "minus" sign in the case of negative integers. The "plus" sign does not appear at all, and the "minus" sign can appear at the expense of one position in the field. Let's add a further modification to the example and see what happens:

```
100 X=PI↑10
110 PRINT X
120 PRINT USING "+6D.2D":X
130 END
```

The output this time is the following:

```
93648.0474761
+93648.05
```

The field width itemizes into:



If X had happened to be negative, then the "+" would be replaced with a "–". The field width would remain at 10. The "–" modifier works in essentially the same way: if the number is negative the "–" appears, otherwise a space appears.

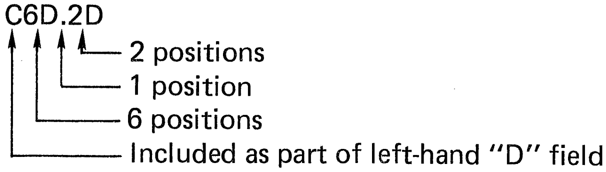
You can cause commas to appear in the output to the left of the decimal point by invoking the "C" modifier:

```
100 X=PI↑10
110 PRINT X
120 PRINT USING "C6D.2D":X
130 END
```

The output now looks like the following:

```
 93648.0474761
93,648.05
```

The field width breaks down into



Notice that the "C" modifier is included within the left-hand "D" field. This means that if the number of digits to the left of the decimal, plus commas, plus sign (for negative numbers) exceeds the left hand "n" modifier (6 in this case), then a field overflow occurs. As it happens, the number we have been working with in these past examples has 5 digits to the left of the decimal; the comma occupies the 6th position. Let's change the sign of X to cause a field overflow and see what happens:

```
100 X=-(PI↑10)
110 PRINT X
120 PRINT USING "C6D.2D":X
130 END
```

Here is the result:

```
 -93648.0474761

FIELD OVERFLOW IN PRINT FORMAT IN LINE 120 - MESSAGE NUMBER 87
```

The overflow occurred because the machine attempted to place 7 items in a field of 6 positions. The minus sign is the 7th item.

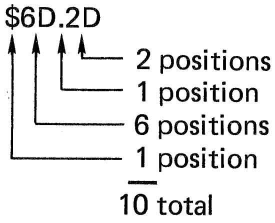
If you want your output to assume a "dollars and cents" appearance, this is easily accomplished through use of the "$" modifier:

```
100 X=PI↑10
110 PRINT X
120 PRINT USING "$6D.2D":X
130 END
```

The output:

```
93648.0474761
$93648.05
```

The "$" modifier differs from "C" in that the dollar sign is not part of the left-hand "D" field. The width of the field will therefore break down into the following:



Now, let's try another run of the same example, this time specifying a dollar sign, commas, and a plus sign:

```
100 X=PI↑10
110 PRINT X
120 PRINT USING "$+C6D.2D":X
130 END
```

The result:

```
 93648.0474761
$+93,648.05
```

There is one additional modifier associated with the "D" field operator: the "F" modifier. This modifier works with "D" similar to the way it works with "A": it results in a field big enough to contain the information. "FD" will round the number at the decimal point and output all significant digits as an integer. A minus sign will precede the output if the number is negative. "FD.FD" will output the number in essentially the same fashion as a normal

PRINT statement, except, of course, PRINT by itself gives you no control over location of the output. "$CFD.FD" will give you a leading dollar sign plus comma separators inserted where needed.

Example:

```
100 X=PI↑10
110 PRINT X
120 PRINT USING "$CFD.FD":X
130 END
```

Results:

```
 93648.0474761
$93,648.0474761
```

There is a restriction on the use of $, + or –, and C. The restriction is that if they are used, they must appear in the following order:

1. $
2. + or –
3. C

Examples are "$+C6D.2D" and "$CFD.2D".

The following is a quick summary of the "D" field operator and its associated modifiers:

- "5D" will output a right justified number as an integer, rounded to the left of the decimal. The field will be 5 characters wide. If the number is negative, the minus sign will take one place in the field. Leftmost unused positions will be padded with spaces.

- "–5D" specifies an output field of 6 characters. This is similar to "5D" except there is one "floating" minus sign or, if the number is positive, one padded space.

- "+5D" also specifies an output field of 6 characters. There is a "floating" plus sign or, if the number is negative, a floating minus sign. (The "+" modifier gives you a plus sign with positive numbers and a minus sign with negative numbers. The "–" modifier gives you a minus sign if the number is negative, and a space if it is positive.)

- "–5D." calls for an output field of 7 characters: 5 digits, a minus sign, and a decimal point.

- "+5D.2D" specifies an output field of 9 characters: a plus sign, 5 digits, a decimal point, and two more digits.

- "FD" will round the number at the decimal point and produce an output field large enough to contain all digits as an integer. If appropriate, a minus sign will precede the output.

- "$5D" calls for an output field of 6 characters. A dollar sign will precede the output.
- "C5D" generates commas. The number of digits, plus commas, plus sign (for negative numbers) must not exceed the number indicated by the "n" modifier.

Scientific Notation Field Operator — E

The "E" field operator outputs a number as a leading mantissa followed by an exponent, as in . . .

1.23E+010

where the E stands for "exponent". The exponent is the power of 10.

The "E" field operator accepts three modifiers; the "n" modifier, which specifies the maximum number of digits that appear in the mantissa to the right of the decimal point, the "F" modifier which specifies "enough" digits in the mantissa to represent the significant digits present, and the "+" modifier which affects the sign of the mantissa.

The following discussion about the "E" field operator and its modifiers relies largely on examples. These examples originate from the following program which allows you to input a format string at "run" time and observe the formatted output. The familiar value PI (3.14159265359) is used as the number being output.

The program:

```
100 X=PI
110 PRINT "FORMAT STRING= ";
120 INPUT A$
130 PRINT "X WITH NON-FORMATTED OUTPUT=";X
140 PRINT "X WITH ""E"" FORMATTED OUTPUT=";
150 PRINT USING A$:X
160 END
```

The modifier "n" controls the number of digits that appear to the right of the decimal point in the mantissa. If "n" is absent, the default value is 1. This is illustrated by running the previous program and using "E" as the format string:

The results:

```
FORMAT STRING= E
X WITH NON-FORMATTED OUTPUT=3.14159265359
X WITH "E" FORMATTED OUTPUT= 3.1E+000
```

Since "n" defaults to 1, "E" and "1E" produce the same result. When "n" is used, it must be in the range of 1 to 11. Try running the program again, this time setting "n" to 4:

The results:

```
FORMAT STRING= 4E
X WITH NON-FORMATTED OUTPUT=3.14159265359
X WITH "E" FORMATTED OUTPUT= 3.1416E+000
```

Notice that rounding occurs in the mantissa; the mantissa is always rounded at the digit being "pointed to" by "n". A format string "11E" produces output with 19 character maximum length (this includes the leading space).

The results:

```
FORMAT STRING= 11E
X WITH NON-FORMATTED OUTPUT=3.14159265359
X WITH "E" FORMATTED OUTPUT= 3.14159265359E+000
```

The "F" modifier specifies that only the significant digits in the mantissa are to be output. That is, trailing zeros in the mantissa are suppressed. Run the program again using the format string "FE":

The results:

```
FORMAT STRING= FE
X WITH NON-FORMATTED OUTPUT=3.14159265359
X WITH "E" FORMATTED OUTPUT= 3.141592654E+000
```

The "FE" format string causes the number of displayed digits in the mantissa to "range" between 1 and 9 depending on the number being printed. The only way to get 10 or 11 digits in the output is to specify "10E" or "11E".

Now let's modify the value of X in line 100 to reduce the number of significant digits:

```
100 LET X=3.14
```

A format string "FE" now yields the following:

```
FORMAT STRING= FE
X WITH NON-FORMATTED OUTPUT=3.14
X WITH "E" FORMATTED OUTPUT= 3.14E+000
```

The "+" modifier works in essentially the same fashion as it does with the "D" field operator. When the "+" modifier is absent, the sign of the mantissa is as follows:

- For negative numbers, the sign is minus (—).
- For non-negative numbers, including zero, a space precedes the mantissa.

When the "+" modifier is present, the sign of the mantissa is according to the following:

- For negative numbers, the sign is minus (—).
- For zero, a space precedes the mantissa.
- For positive numbers, the sign is plus (+).

To illustrate the effects of the "+" modifier, let's first do another modification of the value of X in line 100:

```
100 LET X=-PI
```

A format string "+E" yields the following:

```
FORMAT STRING= +E
X WITH NON-FORMATTED OUTPUT=-3.14159265359
X WITH "E" FORMATTED OUTPUT=-3.1E+000
```

This is the same as if the format string were simply "E", because, for negative numbers, a minus sign is output regardless of the presence or absence of "+". Notice how the non-formatted output is shifted one place to the right to accommodate the minus sign. The difference becomes apparent with positive numbers:

```
100 LET X=PI
```

The format string "+E" now produces this:

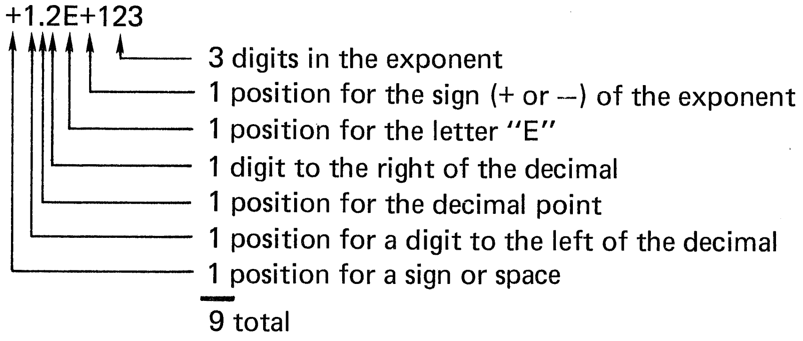
```
FORMAT STRING= +E
X WITH NON-FORMATTED OUTPUT=3.14159265359
X WITH "E" FORMATTED OUTPUT=+3.1E+000
```

Notice the presence of the plus sign.

To summarize the "E" field operator, the form of the specification is:

$$\begin{bmatrix} + \\ - \end{bmatrix} \begin{bmatrix} F \\ n \end{bmatrix} E$$

The minimum field width for the "E" field operator, resulting from a format string like "E" or "1E" or "+1E", is 9 characters:



+1.2E+123

3 digits in the exponent
1 position for the sign (+ or –) of the exponent
1 position for the letter "E"
1 digit to the right of the decimal
1 position for the decimal point
1 position for a digit to the left of the decimal
1 position for a sign or space

9 total

The maximum field width is 19 characters, resulting from a specification like "11E" or "+11E"; 11 digits appear to the right of the decimal point in the mantissa.

When using the "n" modifier, "n" must be in the range 1 through 11. The "F" modifier includes digits to the right of the decimal "as required" to output only the significant digits, up to a maximum of 9 (equivalent to 9E).

Space Field Operator — X

The "X" field operator is used when you want to output spaces (blanks). You might, for example, want to output a number, then 10 spaces, then another number, etc. Accepting only the "n" modifier, and defaulting to "1X" in the absence of a modifier, this field operator is especially easy to use. "10X" outputs 10 spaces, "X" outputs one space, etc. The only restriction is that "n" must be an integer between 1 and 255. This field operator does not require a corresponding item in the PRINT USING variable-list unlike "A", "D", and "E". When the BASIC interpreter encounters an X field operator in a format string, it outputs the specified number of spaces and then goes on. A brief example:

```
100 IMAGE 48(2D,4X)
110 DIM A(48)
120 FOR I=1 TO 48
130 A(I)=I
140 NEXT I
150 PRINT USING 100:A
160 END
```

The output:

```
 1     2     3     4     5     6     7     8     9    10    11    12
13    14    15    16    17    18    19    20    21    22    23    24
25    26    27    28    29    30    31    32    33    34    35    36
37    38    39    40    41    42    43    44    45    46    47    48
```

Literal Field Operator — "

It is possible to include a character string within a format string, and have the BASIC interpreter output the character string as part of the ASCII data string. The field operator that provides this facility is the familiar quotation mark. This operator accepts only the "n" modifier, which must be in the range 1 through 255. The character string which is enclosed within the quote marks is repeated "n" times.

As an example, the following program uses the characters "|" and "—" to produce vertical and horizontal dashed lines to organize the tabular output consisting of powers and roots for the first 20 positive integers. The dashed lines are produced by placing the two characters (" |" and "—") within the format string. The program:

```
100 GOSUB 200
110 PRINT USING 120:" NUMBER","SQUARED","  CUBED","SQR. RT","CUBE RT"
120 IMAGE 5(" | ",7A), " | "
130 GOSUB 200
140 FOR I=1 TO 20
150 PRINT USING 160:I,I↑2,I↑3,I↑(1/2),I↑(1/3)
160 IMAGE 3(" | ",7D), 2(" | ",3D.3D), " | "
170 NEXT I
180 GOSUB 200
190 GO TO 220
200 PRINT USING "X,51""-""":
210 RETURN
220 END
```

The output results:

```
 ---------------------------------------------------
 |  NUMBER | SQUARED |   CUBED | SQR. RT | CUBE RT |
 ---------------------------------------------------
 |       1 |       1 |       1 |   1.000 |   1.000 |
 |       2 |       4 |       8 |   1.414 |   1.260 |
 |       3 |       9 |      27 |   1.732 |   1.442 |
 |       4 |      16 |      64 |   2.000 |   1.587 |
 |       5 |      25 |     125 |   2.236 |   1.710 |
 |       6 |      36 |     216 |   2.449 |   1.817 |
 |       7 |      49 |     343 |   2.646 |   1.913 |
 |       8 |      64 |     512 |   2.828 |   2.000 |
 |       9 |      81 |     729 |   3.000 |   2.080 |
 |      10 |     100 |    1000 |   3.162 |   2.154 |
 |      11 |     121 |    1331 |   3.317 |   2.224 |
 |      12 |     144 |    1728 |   3.464 |   2.289 |
 |      13 |     169 |    2197 |   3.606 |   2.351 |
 |      14 |     196 |    2744 |   3.742 |   2.410 |
 |      15 |     225 |    3375 |   3.873 |   2.466 |
 |      16 |     256 |    4096 |   4.000 |   2.520 |
 |      17 |     289 |    4913 |   4.123 |   2.571 |
 |      18 |     324 |    5832 |   4.243 |   2.621 |
 |      19 |     361 |    6859 |   4.359 |   2.668 |
 |      20 |     400 |    8000 |   4.472 |   2.714 |
 ---------------------------------------------------
```

Line 100 directs the flow of program execution to a subroutine beginning at line 200. This subroutine first skips one space (outputs a blank), and then outputs a row of minus signs before returning to line 110. There, the column headings are printed out. To use the repeat field operator, and to therefore make the format string easier to write, the 5 column headings are the same length: 7 characters, with spaces inserted where necessary to bring the length up to 7. The format string in line 120 causes a vertical bar followed by a column heading to appear 5 times; the trailing "|" provides the 6th vertical dashed column. Once this is done, the subroutine in line 200 is again called to output a row of dashed lines. Next, the powers and roots of the first 20 integers are output by a loop beginning in line 140. The IMAGE statement (line 160) again contains character strings within the format string: the characters " | " (a space, a vertical bar, a space). Notice how the "n" modifiers for the "D" field operators are chosen to produce numeric output fields with a width of 7 characters to correspond to the width of the column headings. Finally, the subroutine in line 200 is called a third time to produce the bottom dashed line.

It is evident from the above example that character string output is generated by enclosing the character string in single quotation marks for format specifications in an IMAGE statement (as in lines 120 and 160), and double quotes for format specifications assigned to string variables and used directly in the PRINT USING statement (line 200). If the output character string contains quote marks, then they must be double for format specifications in IMAGE statements and quadruple for format specifications assigned to string variables and used directly in the PRINT USING statement. For example, the following statements contain a character string that is to be output; the string contains quotation marks. Note the use of quote marks:

```
100 PRINT USING 110:
110 IMAGE 10X,"A ""CHARACTER"" STRING"
120 END


RUN
          A "CHARACTER" STRING
```

In contrast, the following statements produce output identical to the preceding statements; the difference is that this time a format is specified directly in the PRINT USING statement.

```
100 PRINT USING "10X,""A """"CHARACTER"""" STRING""":
110 END

RUN
          A "CHARACTER" STRING
```

Again, notice the quotation marks. The first and last quotation marks delimit the format specification. The character string being output is enclosed within double quotes. The quadruple quotes result in the generation of single quotes in the output. In all cases, there should be an even number of quote marks present.

Page Field Operator — P

The "P" operator places a PAGE command (CTRL L) in the ASCII data string which homes the cursor and clears the screen. The "P" field operator accepts an "n" modifier; "nP" generates n PAGE's. If you want output to begin at the upper left corner of the screen, this can be done with the "P" operator. (This can also be done with a PAGE statement, but at the expense of adding another line to the program).

Line Feed Field Operator — L

The line feed operator, "L", causes the cursor to drop to the next line on the screen. This operator accepts the "n" modifier; nL generates n line feeds. A brief example:

```
100 PRINT USING "10(""LINE FEED"",L)":
110 END
```

```
LINE FEED
         LINE FEED
                  LINE FEED
                           LINE FEED
                                    LINE FEED
                                             LINE FEED
                                                      LINE FEED
                                                               LINE FEED

LINE FEED
         LINE FEED
```

Notice (above) how the output "wraps around" the screen when it comes to the end of a 72 character line.

Carriage Return Field Operator — /

The "/" operator causes a "carriage return" character to be output. On the display, however, this is seen as a "carriage return, line feed" because the screen issues the accompanying "line feed". As an example, suppose you want to generate the following headings:

```
OBS.          % OF          ADJ.
NO.           ERROR         RATIO
```

Here is one approach:

```
100 PRINT USING 110:"OBS.","% OF","ADJ.","NO.","ERROR","RATIO"
110 IMAGE 3(8A),/,3(8A)
120 END
```

```
OBS.    % OF    ADJ.
NO.     ERROR   RATIO
```

The same output can be obtained with the following program, which uses nested parentheses to shorten the IMAGE statement:

```
100 PRINT USING 110:"OBS.","% OF","ADJ.","NO.","ERROR","RATIO"
110 IMAGE 2(3(8A),/)
120 END
```

```
OBS.    % OF    ADJ.
NO.     ERROR   RATIO
```

Tab Field Operator — T

The "T" operator is used in conjunction with the "n" modifier to control the logical position of output. "Logical position" refers to the next position on a line that is to be used.

Whenever a "carriage return" is executed (which is normally the case following the completion of a PRINT statement), the logical position is set to 1. This means that the next character output occupies the 1st position on the line. If you want to make the next item appear in the 30th position on a line, then you specify "30T". An expression of the form "nT" outputs spaces until the logical position n is reached. If the logical position is already n when "nT" is encountered, no spaces are output. If the logical position is already greater than n, an error results and program execution is aborted.

As a brief example, let's assume that you want to generate the headings "SUN, MON, TUE", etc., as part of an undertaking. You want "SUN" to start in position 5, "TUE" to start in position 10, "WED" in position 15, etc. With the "T" operator, the method becomes quite straightforward.

The program:

```
100 PRINT USING 110:"SUN","MON","TUE","WED","THU","FRI","SAT"
110 IMAGE 5T,3A,10T,3A,15T,3A,20T,3A,25T,3A,30T,3A,35T,3A
120 END
```

```
SUN  MON  TUE  WED  THU  FRI  SAT
```

Suppress Field Operator — S

Normally, at the end of a PRINT sequence, the BASIC interpreter outputs a "carriage return" character and the display follows with a "line feed". This places the cursor at the beginning of the next line. If you want to alter this condition so that the cursor remains at the end of the just-printed line, this is accomplished with the "S" field operator. One restriction; if you use "S", it must be the last field operator in the format string.

A short example:

```
100 PRINT USING 110:"JAN","FEB","MAR","APR","MAY","JUNE","JULY","AUG"
110 IMAGE 8(6A),S
120 PRINT USING 130:"SEPT","OCT","NOV","DEC"
130 IMAGE 4(6A)
140 END
```

```
JAN   FEB   MAR   APR   MAY   JUNE  JULY  AUG   SEPT  OCT   NOV   DEC
```

This particular program solves the problem of outputting more information than can fit on the same line as the "PRINT USING" statement. That is, one line of output can contain 72 characters, but those same 72 characters can not fit into a program line because of the amount of space taken by the line number and the BASIC statement. The solution is to break the line up into two lines, and use the "S" operator to suppress the "carriage return" that normally follows upon completion of a PRINT statement.

No-Op Field Operator — ,

The comma, as seen in previous field operator discussions, is generally used as a "no-operation" field operator. It has been used primarily to separate various fields for visual legibility, which organizes complex format strings. Spaces can also be used as separators, making the following format strings equivalent:

6 X 3 D "JOHN"/3 (2 X 4 " X "2 D) 5 E L

6 X, 3D, "JOHN" , / , 3 (2 X, 4 " X " , 2 D) , 5 E , L

6 X 3D "JOHN" / 3 (2X 4 "X" 2D) 5E L

6X, 3D, "JOHN", /, 3(2X, 4"X", 2D) 5E, L

There are, however, two exceptions to the "no-op" nature of the comma as a field operator:

(1) With "D" field operators, "6D.3D" is different from "6D.,3D" or "6D. 3D" or (6D.) 3D". The "6D.3D" form specified one field with a 6 digit integer portion, a decimal point, and a 3 digit decimal portion. The other 3 forms all specify two fields; a 6 digit integer with a decimal point, and a 3 digit integer without a decimal point.

(2) With " field operators, "ABC" "DEF" differs from "ABC", "DEF". The first form prints ABC"DEF; the other form prints ABCDEF.

Field Operator Summary

The accompanying table summarizes the field operators and associated modifiers that are available for the construction of format strings.

Field Operator	Description	Modifiers						
		n	F	+	—	.	$	C
A	Character String	X	X					
D	Digit	X	X	X	X	X	X	X
E	Scientific format	X	X	X				
"	Literal	X						
X	Space	X						
T	Tab	X						
/	Carriage return	X						
P	Page	X						
(	Begin repeat	X						
)	End repeat							
L	Line feed	X						
S	Suppress CR							
,	No-op							

DATA FILES

Background Information

The kind of magnetic tape files introduced in the earlier "DIRECTIVES" section were program files. The process for transferring a BASIC program from the random access read/write memory to a magnetic tape file is as follows:

For "NEW" Files	For Existing Files
* Create the file: FIND (*n*) MARK 1, (*length*) * Position the tape at the beginning of the file: FIND (*n*) * Store the program: SAVE	* Position the tape at the beginning of the file: FIND (*n*) * Store the program: SAVE

Where (*n*) is the number of the file that is to contain the program, and (*length*) is the storage requirement in bytes.

The reverse process, that of transferring programs from tape to the Graphic System memory, is:

* Position the tape at the beginning of the file:

  FIND (*n*)

* Initiate the transfer:

  OLD

where (*n*) is, again, the number of the file.

The way you store and retrieve data is conceptually quite similar to the way you store and retrieve programs. The main difference lies in the keywords that are used. To output a program to the tape, you use SAVE. To output data to the tape, you use PRINT or, as will be discussed, WRITE. Programs are retrieved from the tape with OLD; data are retrieved with INPUT or READ.

Since you can store data on tape with PRINT or WRITE and get it back with INPUT or READ, you might have suspected by now that there are two different kinds of data files, and indeed this is the case. PRINT and INPUT deal with ASCII data files, while READ and WRITE pertain to binary files. Remember that the Graphic System works with binary patterns (1's and 0's), and that ASCII refers to a standardized means of encoding the various alphanumeric characters A-Z and 0-9 plus various other special symbols. The ASCII code exists so that devices like the Graphic System can communicate with teletype machines and other ASCII compatible hardware.

ASCII files and binary files differ not only in name. Again, the Graphic System functions internally with binary patterns; it does not do internal operations using ASCII codes. In order to input and output ASCII data, the system has to convert the ASCII to binary, perform the necessary operations in binary, and then convert from binary to ASCII. What this means, in terms of the differences between ASCII and binary files, is that binary data transfers are considerably faster than ASCII transfers. Binary files already exist in internal system notation; ASCII files require time-consuming conversions.

The PRINT/INPUT statements (for ASCII files) and READ/WRITE statements (for binary files) need to be expanded somewhat so the file information can be directed to and from the magnetic tape. Obviously, PRINT by itself will output information to the screen; conversely, INPUT by itself causes the Graphic System to expect data from the keyboard. READ, you will recall, causes the Graphic System to look for data from a DATA statement. What is needed, therefore, is the means to force the System to PRINT and WRITE to the tape, and INPUT and READ from the tape. The way this is done is through the use of a *primary address*. That is, you supply the PRINT (or READ/WRITE/INPUT) statement with the "address" of the magnetic tape.

Addressing Peripherals

The Graphic System treats the magnetic tape as though it is a peripheral device, separated from the processor but linked with communication lines. Similarly, the keyboard is regarded as a peripheral, as is the screen. Also, from the standpoint of being a source of data, the DATA statement is regarded as a peripheral even though it isn't really a "device" in the usual sense. Each one of these devices (mag tape, keyboard, display, and DATA statement) has an assigned *primary address*. These addresses are as follows:

Device	Primary Address
Keyboard	31
Display (screen)	32
Mag tape	33
DATA statement	34

(It should be pointed out that this is not an exhaustive list of primary addresses. Also, there is a group of secondary addresses. The use of these additional addresses is, however, beyond the scope of this discussion. Refer to the Operators Manual.)

The method of storing and retrieving data on the magnetic tape is now beginning to come into focus:

- First, create a data file on the tape of an appropriate size (more on this later). The familiar MARK statement will do this.
- Next, position the tape at the beginning of the file with the FIND statement.
- Now, assuming (arbitrarily) that the data to be stored is in an array "A", an extended version of PRINT will transfer the data to the tape at primary address 33:

```
PRINT @ 33: A
```

- Note the "@" sign which is used to transmit the address, and the colon delimiter. Incidentally, the primary address can be specified with a numeric variable.
- The tape now contains the contents of array "A" in a data file. It is an ASCII file because PRINT only pertains to ASCII files. WRITE @ 33: A would have generated a binary file.

The version of PRINT that you have been using up to now has sent data to the screen for display. That is, a statement like PRINT X causes the current value of X to be output to the screen. This is as though you had said PRINT @ 32: X (the primary address of the screen is 32). As it turns out, PRINT has a default primary address of 32. That is, when you say PRINT, the Graphic System issues the default address of 32. Similarly, INPUT has a default primary address of 31 (the keyboard). READ also has a default –34 (the DATA statement). In fact, all of the I/O statements have default addresses. The device addresses listed earlier are repeated below, followed by a tabulation of the default addresses for various I/O statements:

DEVICE ADDRESSES

Device	Primary Addresses
Keyboard	31
Display	32
Mag tape	33
DATA statement	34

DEFAULT I/O ADDRESSES

Statement	Default Address
INPUT	31
PRINT	32
READ	34
WRITE	33
LIST	32
OLD	33
SAVE	33
MARK	33

Therefore, when you write something like INPUT A$, the System interprets this to be INPUT @ 31: A$, or, to translate, "Input, from the keyboard, A$". Similarly, a statement like READ Y is interpreted automatically as READ @ 34: Y or "Read, from the DATA statement, the value of Y". It is apparent from all this that you have been addressing peripherals all along, but it has not been obvious. It should also be clear that a convenient means exists to cause the Graphic System to input data and output data to and from the device of your choosing: this means is found in the primary address. Further, the subject of data files is now illuminated considerably: to input/output data from the tape, just specify an address of 33 with the appropriate I/O statement. In the paragraphs that follow, the two types of data files — ASCII and binary — are discussed.

ASCII Files

Program files are ASCII files. This means that each character in each line of the program is represented as an ASCII character. Data files, also, can be made into ASCII files. This is done by sending the data to the tape with a PRINT @ 33: statement. The data is recovered with an INPUT @ 33: statement.

As an illustration, let's build a simple program that will store data in an ASCII data file. First, however, we should determine the next available file number on the tape. With the tape in the tape drive, the TLIST statement will yield a directory of the files on the tape similar to the following:

```
TLIST
 1      ASCII   PROGRAM        1024
 2      ASCII   PROGRAM        1024
 3      ASCII   PROGRAM        2560
 4      ASCII   PROGRAM        768
 5      ASCII   PROGRAM        1536
 6      ASCII   PROGRAM        4608
 7      ASCII   PROGRAM        768
 8      ASCII   PROGRAM        1536
 9      ASCII   PROGRAM        3072
 10     LAST                   768
```

With this particular tape, the last file turns out to be file number 10. Now, to generate some data:

```
100 A=1
110 B=2
120 C=3
130 D=4
140 E=5
```

Let's have the file created under program control by making the program prompt for information:

```
150 PRINT "ENTER NEW FILE NUMBER ";
160 INPUT X
170 PRINT "ENTER THE NUMBER OF DATA ITEMS ";
180 INPUT Y
```

The program now has enough information to set up the file. It knows the file number, and it knows the number of data items involved. From the number of data items involved (5 in this case), the amount of storage required can be determined. ASCII files require (and this is important) *18 bytes per numeric data item*. To create the file:

```
190 FIND X
200 MARK 1,18*Y
```

X, remember, is the file number; Y is the number of data items. To store the data on tape:

```
210 FIND X
220 PRINT @33:A;B;C;D;E
```

The program is now able to make an ASCII data file containing the five data items. Notice that semicolons are used as delimiters in line 220. Semicolons should always be used as delimiters when storing data in ASCII files. This ensures maximum density of the data on the tape. There now remains one detail to be taken care of: the file must be closed. That is, the file will be "opened" in order to receive data, and should then be "closed" to prevent further transfers. A file can be closed in a variety of ways. FIND X will close the file and position the tape at the beginning of the file. END closes all files, as does INIT, DELETE ALL, and pressing the BREAK key twice. Also, there is an additional keyword that can be used: CLOSE. CLOSE will close all files. Let's use END:

```
230 PRINT "DATA STORED ON TAPE"
240 END
```

Let's run the program and store the 5 data items. Type RUN and the program responds with:

```
ENTER NEW FILE NUMBER
```

Enter 10 and press RETURN. The program again responds:

```
ENTER NEW FILE NUMBER 10
ENTER THE NUMBER OF DATA ITEMS 5
```

Enter 5 and press RETURN. The program now causes the data to be stored, and responds with:

```
DATA STORED ON TAPE
```

Before writing a short program to recover the data, let's execute a TLIST from the keyboard and see what the directory reveals:

```
TLIST
 1      ASCII   PROGRAM          1024
 2      ASCII   PROGRAM          1024
 3      ASCII   PROGRAM          2560
 4      ASCII   PROGRAM          768
 5      ASCII   PROGRAM          1536
 6      ASCII   PROGRAM          4608
 7      ASCII   PROGRAM          768
 8      ASCII   PROGRAM          1536
 9      ASCII   PROGRAM          3072
 10     ASCII   DATA             768
 11     LAST                     768
```

Notice the appearance of the new ASCII data file that the program created. Let's write another program to recover the data; but first, type in DELETE ALL to clear the System memory. The program:

```
100 FIND 10
110 INPUT @33:A,B,C,D,E
120 PRINT A;B;C;D;E
130 END
```

When you run the program, you get:

```
RUN

 1 2 3 4 5
```

As evidenced by the above output, the data has been successfully recovered from the tape. For the sake of illustration, let's attempt an unsuccessful approach to recover the same data. First, type in DELETE ALL to clear memory. Next, position the tape at the beginning of the file:

```
100 FIND 10
```

Now use two INPUT statements to try to recover the data instead of one:

```
110 INPUT @33:A,B,C
120 PRINT A;B;C
130 INPUT @33:D,E
140 PRINT D;E
150 END
```

When you run this program, here is what you are greeted with:

```
RUN
 1 2 3

EOF ON UNIT 0 IN LINE 130 - MESSAGE NUMBER 48
```

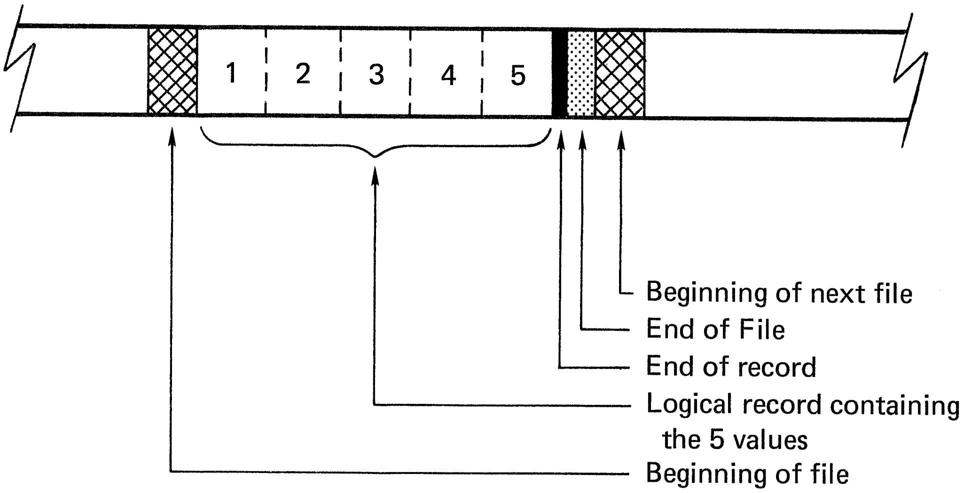
The program recovered the 1, 2, and 3, but it did not recover the 4 and 5. In their place, the error message results. "EOF" means End of File, Unit 0 refers to the magnetic tape.

The reason that the "EOF" error occurred is that ASCII data files are *record-oriented*. Before getting into an elaboration of record-oriented files, let's again examine the statement that caused the Graphic System to place the 5 values on the tape in the data file. The statement was:

```
220 PRINT @33:A;B;C;D;E
```

The variables A,B,C,D, and E contained the values 1, 2, 3, 4, and 5. The statement ends, of course, with a carriage return (the RETURN key). When the system encounters a statement like the one above, it outputs everything between the colon and the terminating carriage return as one unit, or *logical record*.

Conceptually, the tape file looks something like the following after receiving this record:



Now let's examine the statement that successfully retrieved the 5 values:

```
110 INPUT @33:A,B,C,D,E
```

The statement was, of course, preceded with a FIND statement to position the tape at the beginning of the file. When the Graphic System encounters a statement like 110 above, it inputs the entire record. All 5 values are brought in and assigned to the 5 variables listed in the INPUT statement. After the entire record has been input, the tape is situated at the "End of File" position.

With this in mind, let's look at the statements that resulted in the EOF error:

```
110 INPUT @ 33: A,B,C
        .
        .
        .
        .
130 INPUT @ 33: D,E
```

Again, it is understood that a FIND statement positioned the tape at the beginning of the file. The sequence that resulted in the error was as follows:

- When the Graphic System executed the INPUT statement (line 110), the entire record was brought in.
- The first 3 values in the record were assigned to the 3 variables listed with the INPUT statement. The remaining 2 values were not assigned to anything.

- Then, the system was presented with another INPUT statement (line 130). However, the entire logical record containing the 5 values was previously input by line 110. This particular file consists of only one record, and the tape was therefore already situated at the "End of File" position. The error message resulted.

The discussion this far has centered around numeric data. Character strings, too, can be stored in ASCII files. There is an additional detail to be taken care of, however, when storing strings in this manner; a carriage return (the RETURN key) must be used to terminate all strings. With ASCII files, the system requires a carriage return at the end of strings. And, you will recall, the carriage return constitutes an "End of Record."

To illustrate, suppose that you want to store four strings as assigned to string variables below:

```
100 A$="AARDVARK"
110 B$="ABACUS"
120 C$="ABALONE"
130 D$="ABANDON"
```

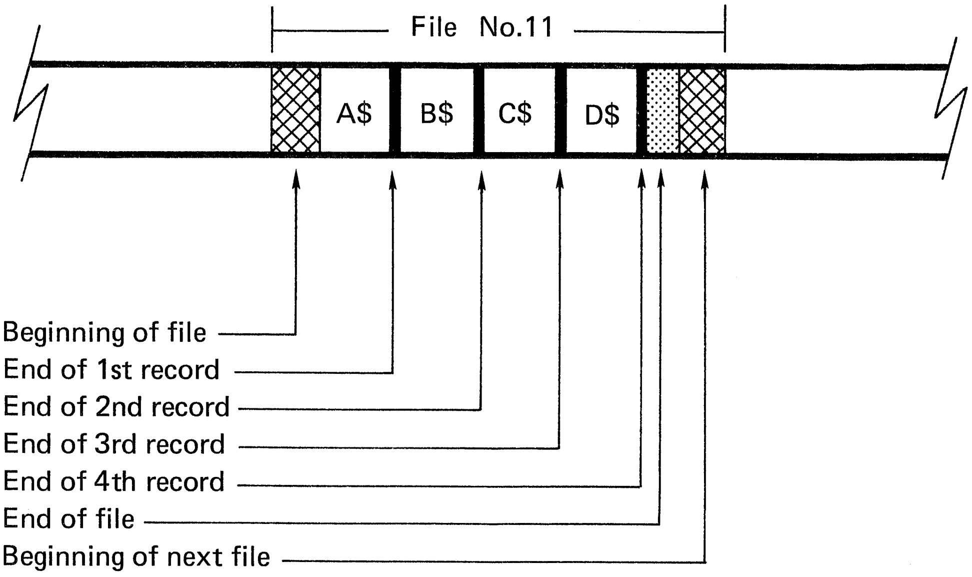
Well, a file needs to be created; let's use file number 11. The space requirement will be 1 byte per character. Instead of counting each character, the Graphic System can do that for us:

```
140 FIND 11
150 MARK 1,LEN(A$)+LEN(B$)+LEN(C$)+LEN(D$)
```

We now have a file of sufficient size to contain the string data. Remembering that the system requires a carriage return at the end of strings, the following statements are indicated:

```
160 FIND 11
170 PRINT @33:A$
180 PRINT @33:B$
190 PRINT @33:C$
200 PRINT @33:D$
210 END
```

The file "looks" like the following:



The file contains 4 records, one for each string variable. The procedure for recovery of the string data contains no surprises:

```
100 FIND 11
110 INPUT @33:A$,B$,C$,D$
```

The first record is input and assigned to A$; the second is assigned to B$, etc.

If you want to intermix string and numeric data in an ASCII file, you should take care to ensure that every transition from numeric data to string data (or string data to numeric) is separated with a carriage return.

Arrays are stored on tape in row-major order and recovered in the same way. Allocate space on the tape by figuring 18 bytes per data item.

To conclude this discussion about ASCII data files, the following points should be remembered:

(1) ASCII data files are *record-oriented*. This means that the smallest working unit is the record, not individual data items. A record at a time is both input and output.

(2) ASCII files are comparatively slow. Within the Graphic System, the data exists in binary form. Therefore, when data is input or output to or from an ASCII file, it must undergo time-consuming conversions.

(3) The semicolon should be used as the delimiter between numeric variables in the PRINT statement, and a carriage return must terminate all strings.

(4) ASCII files exist to provide systems compatibility with other ASCII devices.

(5) The space requirements are:

18 bytes per numeric data item

1 byte per character per string data item

Binary Files

Binary files accomplish essentially the same thing as ASCII files — they let you store and retrieve data. With binary files, however, you are not presented with the constraints of speed and record-orientation. Except for the fact that different keywords are involved (WRITE and READ versus PRINT and INPUT), you use binary files in pretty much the same way as ASCII files. That is, the files are created with MARK, and the tape is positioned with FIND. Binary files are closed the same way that ASCII files are closed — with FIND, CLOSE, or END.

The following program is essentially a duplication of the earlier program that stored 5 values in an ASCII data file, except this time a binary file is used:

```
100 A=1
110 B=2
120 C=3
130 D=4
140 E=5
150 PRINT "ENTER FILE NUMBER: ";
160 INPUT X
170 PRINT "ENTER NUMBER OF DATA ITEMS: ";
180 INPUT Y
190 FIND X
200 MARK 1,10*Y
210 FIND X
220 WRITE A,B,C,D,E
230 PRINT "DATA STORED ON TAPE"
240 END
```

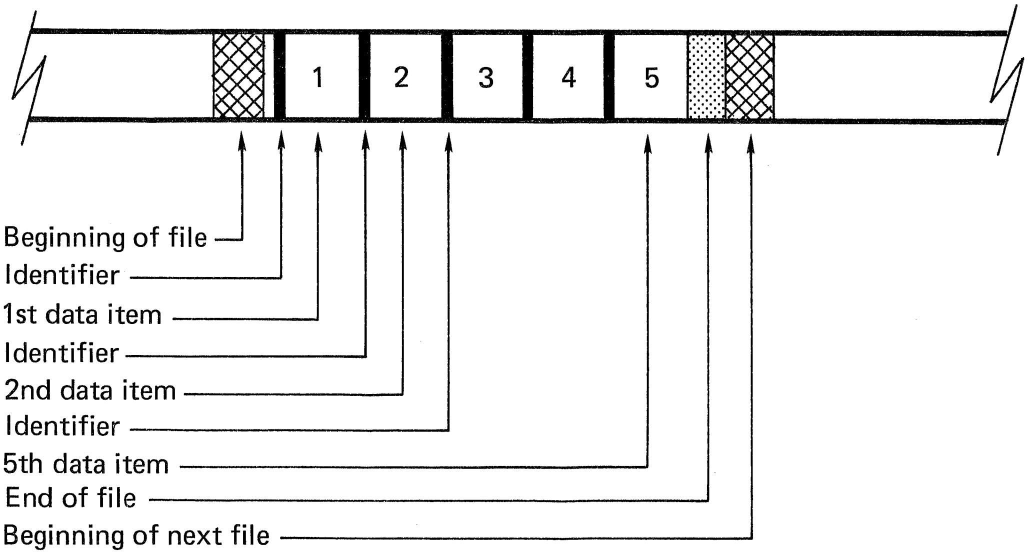
When you run the above program and respond to the promptings (lines 150 and 170), the resulting output on the display looks like this:

```
ENTER FILE NUMBER: 11
ENTER NUMBER OF DATA ITEMS: 5
DATA STORED ON TAPE
```

Execution of a TLIST statement reveals the presence of the newly – formed binary file:

```
1       ASCII   PROGRAM         1024
2       ASCII   PROGRAM         1024
3       ASCII   PROGRAM         2560
4       ASCII   PROGRAM         768
5       ASCII   PROGRAM         1536
6       ASCII   PROGRAM         4608
7       ASCII   PROGRAM         768
8       ASCII   PROGRAM         1536
9       ASCII   PROGRAM         3072
10      ASCII   DATA            768
11      BINARY  DATA            768
12      LAST                    768
```

Notice, in line 220 of the program, the WRITE statement. No special delimiters are required; just the conventional comma is used between data items. The file "looks" like the following:



Note, from the illustration, that each data item carries with it an identifier. This identifier informs the system of the type of data (numeric or string) and its length. Note also the absence of records. Binary files are data item oriented rather than record-oriented.

Line 200 of the program contains a clue as to the space requirements for numeric data in a binary file: 10 bytes per numeric data item. In ASCII files, you will recall, it was 18 bytes.

Binary data is recovered from the tape with the READ statement:

```
100 PRINT "ENTER FILE NUMBER: ";
110 INPUT X
120 FIND X
130 READ @33:A,B,C,D,E
140 PRINT A;B;C;D;E
150 END

RUN
ENTER FILE NUMBER: 11
 1 2 3 4 5
```

These five data items could also be recovered with:

```
100 PRINT "ENTER FILE NUMBER: ";
110 INPUT X
120 FIND X
130 READ @33:A,B,C
140 PRINT A;B;C;
150 READ @33:D,E
160 PRINT D;E
170 END

RUN
ENTER FILE NUMBER: 11
 1 2 3 4 5
```

This was not possible with ASCII files. With binary files, you can input a data item at a time, rather than having to input a record at a time.

With strings, the procedure isn't much different:

```
100 A$="ZAIBATSU"
110 B$="ZANY"
120 C$="ZEALOT"
130 D$="ZEALOUS"
140 Y=LEN(A$)+2+LEN(B$)+2+LEN(C$)+2+LEN(D$)+2
150 PRINT "ENTER FILE NUMBER: ";
160 INPUT X
170 FIND X
180 MARK 1,Y
190 FIND X
200 WRITE A$,B$,C$,D$
210 END
```

The space requirement for strings is LEN + 2 bytes for each string where LEN is the length of the string. This space requirement is calculated in line 140 and used in line 180. Recovery of the data is straightforward:

```
100 PRINT "ENTER FILE NUMBER: ";
110 INPUT X
120 FIND X
130 READ @33:A$,B$,C$,D$
140 PRINT USING "2/,4(FA,/)":A$,B$,C$,D$
150 END

RUN
ENTER FILE NUMBER: 12


ZAIBATSU
ZANY
ZEALOT
ZEALOUS
```

Again, you can read data items individually:

```
100 PRINT "ENTER FILE NUMBER: ";
110 INPUT X
120 FIND X
130 READ @33:A$
140 PRINT A$
150 READ @33:B$
160 PRINT B$
170 READ @33:C$
180 PRINT C$
190 READ @33:D$
200 PRINT D$
210 END

RUN
ENTER FILE NUMBER: 12
ZAIBATSU
ZANY
ZEALOT
ZEALOUS
```

With binary files, you can intermix data item types without a great deal of concern about delimiters, as in WRITE @ 33: A,B,A$,C,D,B$. The reason this is possible is that each data item carries with it an identifier that lets the Graphic System know what is going on.

Arrays are written on the tape in row major order and recovered the same way. If the array is named "A", you just say WRITE @ 33: A.

The following points conclude the discussion about binary files:

(1) Binary files are data item oriented so that you can read individual data items.

(2) Binary file transfers are faster because the binary to ASCII and ASCII to binary conversions are not necessary.

(3) No special delimiters are needed. The comma will suffice. Numeric and string data can be intermixed with no problem.

(4) The space requirements are:

10 bytes per numeric data item
LEN + 2 bytes per string data item

Determination of Data Type

Occasionally the situation may arise where you lose track of the structure of a data file. The TLIST directive can be used to readily determine the kind of file, but it provides no clue as to the organization of a file. The file might contain string data, numeric data, or a mixture of the two. Unless you can determine the type of data (string or numeric) of the "next" data item in the file, you don't know whether to attempt to assign the item to a string variable or a numeric variable.

This version of BASIC provides the means of examining the "next" data item in a file to determine its type. This means is given by the TYP (for type) function:

```
[Line number] TYP (0)
```

The 0 refers to the magnetic tape. The function will return a number in the range of 0 through 4:

Number	Meaning
0	File not open
1	End of file
2	ASCII file
3	Binary numeric data
4	Binary string data

The TYP function is quite useful in conjunction with a GOTO . . . OF . . . or GOSUB. . . OF. . . statement, as in

```
100 FIND N
110 GO TO TYP(0)+1 OF 500,600,700,800,900
```

This enables you to route program control to an appropriate routine to handle the various data types. Notice from line 110 above that the value of 1 is added to the number returned by the TYP function because TYP can return a 0. A "GOTO 0" or "GOSUB 0", you will recall, is ignored.

The following program includes an example usage of TYP. The program will extract and print on the display the contents of a data file, either ASCII or binary.

```
100 PRINT "ENTER FILE NUMBER: ";
110 INPUT N
120 FIND N
130 PRINT "CONTENTS OF FILE:"
140 PRINT
150 GO TO TYP(0)+1 OF 390,170,220,310,350
160 REMARK ROUTINE FOR "END OF FILE"
170 PRINT
180 PRINT
190 PRINT "END OF FILE"
200 GO TO 390
210 REMARK ROUTINE FOR ASCII FILES
220 PRINT "ASCII FILE"
230 PRINT "ENTER FILE SIZE IN BYTES: ";
240 INPUT B
250 DELETE B$
260 DIM B$(B)
270 INPUT @33:B$
280 PRINT B$
290 GO TO 150
300 REMARK ROUTINE FOR BINARY FILE NUMERIC ITEMS
310 READ @33:A
320 PRINT A;" ";
330 GO TO 150
340 REMARK ROUTINE FOR BINARY FILE STRING ITEMS
350 READ @33:A$
360 A$=A$&"  "
370 PRINT A$;
380 GO TO 150
390 END
```

The file is opened and the tape is positioned at the beginning of the file with the FIND statement in line 120. The TYPE function in line 150 routes control to one of 5 routines.

The "End of File" routine (lines 170—200) simply prints a message and then transfers control to the END statement. ASCII files are handled with the routine of lines 220—290. This routine prints a message to identify the file as an ASCII file, and then goes on to ask for the size of the file in bytes. This number can be obtained from a TLIST printout, and is used to dimension a string variable large enough to contain the entire file. Recall that ASCII files are record oriented; the entire file could be one logical record, or it could be several logical records. In either case, the "next" logical record is input into the string variable and then printed on the screen. From there, control is passed back to line 150, where the TYP function will continue to pass control back to the ASCII routine until the end of the file is reached.

The routine to handle binary file numeric items (lines 310—330) reads one data item at a time. After the data item is read in, it is printed on the display together with a trailing space for readability. Then, control passes back to line 150. Binary string items are handled similarly. A space is concatenated on the end of the string for readability.

File Security

If the need arises to have a program kept secret, this can be done with the following statement:

```
[Line number] SECRET
```

When the SECRET statement is executed, either under program control or from the keyboard, the program currently located in the system memory is marked as being secret. When the program is stored on tape, it can still be loaded into memory with OLD or APPEND, and executed with RUN, but it can not be listed. If you append a secret program file into a non-secret program, then the non-secret program is made secret. This is to preserve the "secret" nature of the appended file. A secret file also can not be loaded into memory and then stored in a different file on the tape, because that amounts to an attempt to "list" the secret program on tape. SECRET files are just that: they are secret, even from yourself.

The following dummy program demonstrates the use of SECRET.

```
100 REM ****DUMMY PROGRAM USING "SECRET"****
110 REM
120 REM
130 REM
140 REM
150 REM
160 REM
170 REM
180 REM
190 REM
200 SECRET
210 PRINT "FILE NUMBER: ";
220 INPUT N
230 FIND N
240 MARK 1,SPACE
250 FIND N
260 SAVE
270 END
RUN
FILE NUMBER: 13
```

The program is made secret by line 200. The next four lines store the program on tape. Now, to attempt to recover the program from the tape and list it:

```
FIND 13
OLD
LIST

PROGRAM IS SECRET IN IMMEDIATE LINE - MESSAGE NUMBER 38
```

As is evident from the above, it can not be listed. Notice the manner in which secret files are represented following execution of a TLIST:

```
TLIST
 1      ASCII   PROGRAM          1024
 2      ASCII   PROGRAM          1024
 3      ASCII   PROGRAM          2560
 4      ASCII   PROGRAM          768
 5      ASCII   PROGRAM          1536
 6      ASCII   PROGRAM          4608
 7      ASCII   PROGRAM          768
 8      ASCII   PROGRAM          1536
 9      ASCII   PROGRAM          3072
 10     ASCII   DATA             768
 11     BINARY  DATA             768
 12     BINARY  DATA             768
 13     ASCII   PROGRAM   SECRET 1536
 14     LAST                     768
```

INTERNAL INTERRUPTS

General

An interrupt is a recoverable break in the normal flow of program execution such that flow can be resumed from the point of interruption at a later time. For purposes of this discussion, interrupts occur in two categories: external and internal. An external interrupt is caused by an external device (like a disc or a data acquisition device); internal interrupts are caused by "internal devices" like the tape and screen. This discussion covers internal interrupts only.

A good example of an internal interrupt is one which you may have already experienced — when the screen is full, normal execution stops until you press the PAGE key. In this case, the interrupt is caused by a FULL condition, and processing does not resume until some action (like pressing PAGE) is taken. The Graphic System allows you to write program segments, called "service routines", that will cause the necessary action to take place under program control to satisfy the interrupt. These service routines closely resemble subroutines.

There are three kinds of internal interrupts you can work with:

(1) FULL condition, caused by a full page.

(2) SIZE condition, caused by some arithmetic operation which results in numeric values beyond the range of the machine (like dividing by zero).

(3) EOF (End of File) condition, caused by an I/O attempt which goes beyond the end of file.

In the absence of an appropriate service routine, the occurrence of any of these three conditions (FULL, SIZE, and EOF) will cause the System to terminate execution. With the SIZE and EOF interrupts, an appropriate error message is normally issued; with FULL a small flashing "F" appears in the upper left corner of the screen. Most of the time it is acceptable for the System to simply stop execution when one of these conditions occur. However, the facility does exist whereby you can take appropriate action to service the interrupt under program control, thereby avoiding termination of execution.

This interrupt handling facility is enabled with a pair of BASIC statements. One statement "turns on" this capability, the other statement "turns off" this capability. The two statements are:

```
Line number ON condition THEN line number          [Line number] OFF [condition]
```

The condition can be SIZE or EOF. (FULL is handled differently, using an extended PRINT statement.) Note that the ON statement requires a beginning line number. With the OFF statement, the line number is optional like most BASIC statements. The second line number in the ON statement is the beginning line number of the service routine.

A complete ON statement could therefore be something like

```
100 ON SIZE THEN 1000
```

With a statement like this one, the occurrence of a SIZE error causes control to be transferred to line 1000, as though you had executed a GOSUB 1000. From there, your service routine would take some action to correct the SIZE condition, such as altering the value of a variable to bring it back within the range of the machine. If the routine ends with a RETURN statement, control will be returned to the line number following the one that resulted in the SIZE condition.

Similarly, an ON statement could be

```
200 ON EOF(0) THEN 500
```

If an end-of-file condition arises, then control will be passed to the service routine at statement 500. The zero in "EOF (0)" refers to the magnetic tape.

You disable an ON condition through use of the OFF statement. For instance, if you want to disable a previously enabled ON SIZE, you just include a line like

```
300 OFF SIZE
```

Once control passes line 300, size errors will result in termination of the program. If you want to disable all interrupt handling capabilities, you just enter a line like

```
500 OFF
```

An INIT statement, you will recall, also disables all interrupts.

SIZE Condition

A SIZE condition, and resulting interrupt, occurs whenever a mathematical computation results in a numeric value which is beyond the numeric range of the machine. For example, a divide-by-zero operation normally terminates program execution. In situations where you don't want program execution to halt due to a SIZE condition, you have the alternative to include a service routine in the program to handle the interrupt. This service routine could reset the variable that resulted in the condition, or it could simply print a message on the screen. The ON SIZE statement can also be used to route program flow to a separate set of calculations to be performed in the event of a zero-divide.

The following program is purposely written to cause a divide-by-zero situation and resulting SIZE condition. The program uses a FOR loop to fill an 11 element array. The values assigned to the array are obtained by dividing the value of the FOR variable by itself; the initial and ending values of the FOR variable have been chosen so that it passes through zero, thereby causing a divide-by-zero situation.

```
110 DIM A(11)
120 K=0
130 FOR I=-5 TO 5
140 K=K+1
150 A(K)=I/I
160 NEXT I
170 PRINT A;
180 END

RUN

SIZE ERROR IN LINE 150 - MESSAGE NUMBER 2
```

Note that when the program attempted to run, the only output obtained was the SIZE ERROR message. Control never reached the PRINT in line 170 which would have output the contents of array A. This can be remedied by modifying the program to include an ON. . THEN. . . statement to catch the SIZE condition, and an appropriate service routine to handle the interrupt. The program is repeated below, with the modifications indicated.

```
100 ON SIZE THEN 190
110 DIM A(11)
120 K=0
130 FOR I=-5 TO 5
140 K=K+1
150 A(K)=I/I
160 NEXT I
170 PRINT A;
180 GO TO 220
190 A(K)=0
200 PRINT "SIZE ERROR OCCURED"
210 RETURN
220 END
```

The ON. . . THEN. . . statement directs the flow of control to line 190 in the event of a SIZE condition. As evidenced by the previous attempt to run the program, a SIZE condition will occur in line 150 when the program attempts to divide by zero. When this happens, control will pass to line 190 as directed by the ON. . . THEN. . . statement. Once control reaches statement 190, A(K) is set to zero for visibility in the output, and a message is generated by line 200. The RETURN then passes control back to line 160, which is the line after the one that causes the interrupt. The GOTO in line 180 prevents accidental entry into the service routine.

The output of the modified program is below. Notice the 0 supplied by the service routine in the output from the array.

```
RUN
SIZE ERROR OCCURED

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

EOF Condition

An end of file condition can arise two ways: either from an attempt to read beyond the end of file, or an attempt to write beyond the end of file. Service routines can be written to handle either case. The program that follows attempts to place more data in a file than the file can contain. A FOR loop is used to fill a 100 element array with data; the amount of storage space required to contain 100 data items in a binary file is 1000 bytes. The MARK statement intentionally creates a file too small so that an EOF interrupt will occur when the program runs.

```
100 DIM X(100)
110 FOR T=1 TO 100
120 X(T)=T
130 NEXT T
140 PRINT "FILE NUMBER: ";
150 INPUT N
160 FIND N
170 MARK 1,50
180 FIND N
190 WRITE X
200 END

RUN
FILE NUMBER: 14

EOF ON UNIT 0 IN LINE 190 - MESSAGE NUMBER 48
```

The addition of an ON. . . THEN. . . statement and a service routine handles the interrupt and prevents termination of the program. The modified program appears below with the added statements emphasized.

```
100 ON EOF (0) THEN 220
110 DIM X(100)
120 FOR T=1 TO 100
130 X(T)=T
140 NEXT T
150 PRINT "FILE NUMBER: ";
160 INPUT N
170 FIND N
180 MARK 1,50
190 FIND N
200 WRITE X
210 GO TO 260
220 FIND N
230 MARK 1,1000
```

```
240 PRINT "END-OF-FILE OCCURED; FILE EXPANDED"
250 GO TO 190
260 END

RUN
FILE NUMBER: 14
END-OF-FILE OCCURED; FILE EXPANDED
```

When the EOF condition arises in line 200, control is transferred to line 220 because of the ON. . . THEN. . . in line 100. Once control reaches line 220, the service routine goes on to expand the size of the file and print a message. Then, control goes back to line 190, where the FIND/WRITE sequence is repeated. Line 210 keeps control from again reaching the service routine. Note that a RETURN is not used in this case; program control exits the routine by means of a GOTO.

The above discussion centers around an EOF situation arising from an attempt to write information beyond the end of a file. The reverse situation, that of attempting to read information beyond the end of a file, is discussed below.

The first thing that needs to be done is to generate some data and store it in a file:

```
100 DIM A(5)
110 FOR I=1 TO 5
120 A(I)=I
130 NEXT I
140 PRINT "FILE NUMBER: ";
150 INPUT N
160 FIND N
170 WRITE A
180 END
```

After the 5 items in array A are stored in the file, the END statement closes the file and places an EOF mark after the 5th item. Now, another program is needed that will try to read past the end of the file. Such a program, and the resulting output, appears below:

```
100 DIM A(6)
110 PRINT "FILE NUMBER: ";
120 INPUT N
130 FIND N
140 FOR I=1 TO 6
150 READ @33:A(I)
160 NEXT I
170 PRINT A;
180 END

RUN
FILE NUMBER: 12

EOF ON UNIT 0 IN LINE 150 - MESSAGE NUMBER 48
```

Notice that control never reached line 170.

A modified version of the above program, including a short service routine, appears below followed by the output that is generated.

```
100 DIM A(6)
110 A=0
120 ON EOF (0) THEN 200
130 PRINT "FILE NUMBER: ";
140 INPUT N
150 FIND N
160 FOR I=1 TO 6
170 READ @33:A(I)
180 NEXT I
190 GO TO 220
200 PRINT A;
210 PRINT "END-OF-FILE FOLLOWING DATA ITEM ";I-1
220 END

RUN
FILE NUMBER: 12

 1 2 3 4 5 0


END-OF-FILE FOLLOWING DATA ITEM 5
```

The service routine begins at line 200; all it does is output the contents of array A and then display a message. The array is initialized in line 110; this is done because A is dimensioned to be 6 elements and only 5 elements are read in. The 6th element would be undefined without the initialization, and the PRINT in line 200 would otherwise cause an UNDEFINED VARIABLE error.

FULL Condition

An extended function ROM accessory is required before you can handle "page full" interrupts with a service routine. In lieu of that, however, some measure of control over the "page full" condition can be obtained through an extended version of the PRINT statement. The syntax form of this statement is:

```
[Line number] PRINT @ 32,26: numeric expression
```

The 32 following the "@" symbol is the primary address of the display, and the 26 is a secondary address establishing a "page full mode". Different values for the numeric expression will cause the Graphic System to take specific action when a "page full" situation arises. The various actions that will occur with different values for n are tabulated below.

n	ACTION
0	Normal
1	HOME
2	PAGE
3	COPY & PAGE

Thus, the statement:

```
150 PRINT @32,26:3
```

will cause the system to PAGE the screen when it is full. Likewise, the statement

```
150 PRINT @32,26:2
```

will cause the system to generate a hard copy of the screen if an optional Hard Copy device is attached, followed by a PAGE. Specifying a value of 0 for n will restore the normal "page full" action: execution will terminate when the page is full. A value of 1 will cause the system to HOME when the page is full and continue on, over-writing the information already present on the screen.

SUMMARY

Extensive control of output formatting is made possible with PRINT USING and a format string. The format string can appear as a string enclosed within quotes as part of the PRINT USING statement, or it can appear elsewhere in the program as an IMAGE statement. Either way, the characters in the format string determine the appearance of the output. There are thirteen field operators that determine the output format; most of these operators accept modifiers.

Data is stored in files in a way that is quite similar to the way that programs are stored in files. The main difference lies in the keywords that apply. Instead of SAVE, you use PRINT or WRITE; instead of OLD, you use INPUT or READ. The PRINT, INPUT, and READ statements require inclusion of a primary address of 33 in order to effect data transfers to and from the tape.

Data is stored in ASCII files with the PRINT @ 33: statement and retrieved with the INPUT @ 33 statement. ASCII data files are record-oriented. That is, the smallest unit that can input or output is a record. Records are delimited with a carriage return character. ASCII files are comparatively slow because of the binary/ASCII conversions that must take place. A semicolon is the appropriate delimiter between numeric variables; a carriage return should delimit strings. The storage requirements are 18 bytes per numeric data item and one byte per character (in the case of strings).

Data I/O with binary files is done with WRITE and READ @ 33: statements. Binary files do not require any special delimiters; commas between data items are sufficient. Binary files are not record-oriented. This means you can read or write single data items; you do not have to work with records. Operations with binary files are faster than those with ASCII files because the information already exists in internal notation and does not require conversion. The storage requirements are 10 bytes per numeric data item and LEN + 2 bytes per string data item.

The TYPE function can be utilized to determine the next data item type. SECRET gives you non-listable files where file security is necessary. Files are closed with END, FIND, or CLOSE.

The internal interrupt conditions normally result in termination of program execution, but you have the facility to recover from these conditions by writing service routines. These service routines resemble subroutines, and are accessed with ON. . . THEN. . . At the conclusion of the service routine, control is returned to the interruption point with RETURN. ON. . . THEN. . . enables this capability, OFF disables it.

EXAMPLE PROGRAMS

TITLE: **Amortization Table**

DESCRIPTION: The program outputs an amortization table when supplied with the following data:

- Principal
- Rate
- Payment
- Beginning data

PROGRAM LISTING:

```
100 REMARK AMORTIZATION TABLE PROGRAM
110 PAGE
120 PRINT "ENTER PRINCIPAL ";
130 INPUT P1
140 PRINT "ENTER RATE ";
150 INPUT R
160 PRINT "ENTER PAYMENT ";
170 INPUT P2
180 PRINT "ENTER BEGINNING DATE (MO,DAY,YEAR) ";
190 INPUT M1,D,Y
200 M2=R/12
210 A=1
220 PRINT
230 PRINT
240 PRINT USING "18T,FA":"***AMORTIZATION TABLE***"
250 PRINT USING "60""-""":
260 PRINT USING 270:"DUE","BEGINNING","INTEREST","PRINCIPAL","ENDING"
270 IMAGE 4T,FA,16T,FA,27T,FA,37T,FA,48T,FA
280 PRINT USING 270:"DATE","BALANCE","PAYMENT","PAYMENT","BALANCE"
290 PRINT USING "60""-""":
300 I=P1*M2
310 P3=P2-I
320 N=P1-P3
330 GO TO A OF 340,400
340 IF P3>0 THEN 390
350 PRINT "PAYMENT SET TOO LOW"
360 PRINT
370 PRINT
380 GO TO 120
390 A=2
400 IF N=>0.005 THEN 430
410 P3=P1
420 N=0
430 PRINT USING 440:M1,"/",D,"/",Y,P1,I,P3,N
440 IMAGE 4T,FD,A,FD,A,FD,16T,$FD.2D,27T,$FD.2D,37T,$FD.2D,48T,$FD.2D
450 IF N<0.005 THEN 540
460 IF M1=12 THEN 490
470 M1=M1+1
480 GO TO 510
490 M1=1
```

```
500 Y=Y+1
510 P1=N
520 GO TO 300
530 REMARK "G" IS THE "BELL" CHARACTER
540 PRINT "GGG"
550 END
```

METHODOLOGY: The output appears in five columns, and relies heavily on the formatting capabilities of the PRINT USING statement.

OPERATING PROCEDURE: Type RUN, press RETURN. The program responds with ENTER PRINCIPAL. Enter the principal and press RETURN. The program then asks ENTER RATE. Enter the interest rate as a decimal (9% = .09) and press RETURN. The program responds with ENTER PAYMENT. Enter the payment and press RETURN. The final prompting is ENTER BEGINNING DATE (month, day, year). Enter the date (as in 3, 15, 75) and press RETURN. The table is then generated. If it happens that the payment is incorrectly entered so that it is less than or equal to the interest, the message PAYMENT SET TOO LOW appears.

OUTPUT SAMPLE:

```
ENTER PRINCIPAL 1000
ENTER RATE .09
ENTER PAYMENT 150
ENTER BEGINNING DATE (MO,DAY,YEAR) 7,15,75

                  ***AMORTIZATION TABLE***
------------------------------------------------------------
   DUE          BEGINNING  INTEREST  PRINCIPAL  ENDING
   DATE         BALANCE    PAYMENT   PAYMENT    BALANCE
------------------------------------------------------------
   7/15/75      $1000.00   $7.50     $142.50    $857.50
   8/15/75      $857.50    $6.43     $143.57    $713.93
   9/15/75      $713.93    $5.35     $144.65    $569.29
   10/15/75     $569.29    $4.27     $145.73    $423.56
   11/15/75     $423.56    $3.18     $146.82    $276.73
   12/15/75     $276.73    $2.08     $147.92    $128.81
   1/15/76      $128.81    $0.97     $128.81    $0.00
```

TITLE: **Program to Store Grades in a Binary File**

DESCRIPTION: This is an interactive program which will store student grades in a binary file. The program creates the file, if necessary, (which is the case the first time the program is used), or utilizes a previously established file. The size of the file is determined by the number of students involved and the number of grades per student.

PROGRAM LISTING:

```
100 REMARK PROGRAM TO STORE GRADES IN A BINARY FILE
110 DIM A$(1)
120 PRINT "DOES THE FILE NEED TO BE CREATED? (Y OR N) ";
130 INPUT A$
140 PRINT "WHAT FILE NUMBER? ";
150 INPUT F
160 PRINT "HOW MANY STUDENTS? ";
170 INPUT N
180 PRINT "HOW MANY GRADES PER STUDENT? ";
190 INPUT M
200 IF A$="Y" THEN 220
210 GO TO 240
220 FIND F
230 MARK 1,30*N+10*N*M
240 FIND F
250 WRITE N,M
260 PRINT "ENTER STUDENT DATA"
270 FOR I=1 TO N
280 PRINT
290 PRINT "STUDENT NAME (";I;"): ";
300 INPUT N$
310 WRITE N$
320 FOR J=1 TO M
330 PRINT "GRADE (";J;"): ";
340 INPUT G
350 WRITE G
360 NEXT J
370 NEXT I
380 PRINT
390 PRINT "** DONE **"
400 END
```

METHODOLOGY: The decision to create a new file or to use an existing file is made by testing a string variable to see if it contains a "Y" (for Yes) or "N" (for No). The data defining the file number, number of students, and number of grades per student is input at run time.

The files are created with enough space to contain a 30 character string for each student name in addition to the actual grades (line 230 of the program). The first two items stored in each file are the number of students and the number of grades per student. This makes it easier to retrieve the data from the file because with these two items it is convenient to determine the amount of data in the file. Also, these two items can be used to control the number of iterations that the READ loop will undergo in the program to retrieve the grades.

OPERATING PROCEDURE: Type RUN, press RETURN. The program responds with "DOES THE FILE NEED TO BE CREATED?" Type Y or N (for Yes or No) and press RETURN. The program then asks "WHAT FILE NUMBER?" Enter the number of the file that is to contain the data and press RETURN. The next two promptings are "HOW MANY STUDENTS?" and "HOW MANY GRADES PER STUDENT?" Enter the appropriate amounts and press RETURN. The program then says "ENTER STUDENT DATA" and you then enter the student names and data.

OUTPUT SAMPLE:

```
RUN
DOES THE FILE NEED TO BE CREATED? (Y OR N) Y
WHAT FILE NUMBER? 1
HOW MANY STUDENTS? 4
HOW MANY GRADES PER STUDENT? 4
ENTER STUDENT DATA

STUDENT NAME (1): STUDENT1
GRADE (1): 78
GRADE (2): 98
GRADE (3): 80
GRADE (4): 75

STUDENT NAME (2): STUDENT2
GRADE (1): 88
GRADE (2): 95
GRADE (3): 82
GRADE (4): 80

STUDENT NAME (3): STUDENT3
GRADE (1): 75
GRADE (2): 70
GRADE (3): 80
GRADE (4): 78

STUDENT NAME (4): STUDENT4
GRADE (1): 90
GRADE (2): 95
GRADE (3): 85
GRADE (4): 90

** DONE **
```

TITLE: **Program to Retrieve Grades From Binary File**

DESCRIPTION: The program retrieves student grades from a binary file, and is intended to be used in conjunction with the previous example program. The routine allows you to list out the names and grades of all the students in the file or, if desired, list out only one specified student name and associated grades.

PROGRAM LISTING:

```
100 REMARK PROGRAM TO RETRIEVE GRADES FROM BINARY FILE
110 N$=""
120 PRINT "WHAT FILE NUMBER? ";
130 INPUT F
140 FIND F
150 READ @33:N,M
160 PRINT "ALL STUDENTS, OR ONE STUDENT? (ENTER ""ALL"" OR ""ONE"") ";
170 INPUT A$
180 IF A$="ALL" THEN 240
190 IF A$<>"ONE" THEN 160
200 REMARK THIS POINT REACHED IF A$="ONE"
210 PRINT "WHICH STUDENT? ";
220 INPUT N$
230 REMARK LOOP TO READ IN INFORMATION FOR N STUDENTS
240 FOR I=1 TO N
250 READ @33:S$
260 GO TO (A$="ALL")+1 OF 270,290
270 IF S$=N$ THEN 290
280 GO TO 320
290 PRINT
300 PRINT S$
310 REMARK LOOP TO READ IN M GRADES PER STUDENT
320 FOR J=1 TO M
330 READ @33:S
340 GO TO (A$="ALL")+1 OF 350,370
350 IF S$=N$ THEN 370
360 GO TO 380
370 PRINT "GRADE (";J;"): ";S
380 NEXT J
390 IF S$=N$ THEN 430
400 NEXT I
410 GO TO (A$="ALL")+1 OF 420,430
420 PRINT "STUDENT NAME NOT FOUND IN FILE"
430 END
```

METHODOLOGY: The first two items read from the file are values for N (the number of students in the file) and M (the number of grades per student). Following this, lines 160 and 170 of the program cause the string variable A$ to receive "ALL" or "ONE". Depending on the contents of A$, the program will either list out the entire file or just list out the portion of the file corresponding to the specified student name. The appropriate flow of execution to handle "ALL" or "ONE" is effected primarily by GO TO . . . OF . . . statements in lines 260, 340, and 410. These three statements contain the logical expression A$ = "ALL" which evaluates to a 1 or 0. If A$ does contain "ALL" then the logical expression is true and therefore equal to 1. This 1 is summed with the constant 1, and control is passed to the second statement number specified in the GOTO . . . OF . . . statement. If A$ contains "ONE" (or does not contain "ALL") then the logical expression is 0 (false). The 0 is summed with 1, and control is passed to the first statement number.

The program contains a test in lines 180 and 190 to ensure that only "ALL" or "ONE" is entered into A$.

OPERATING PROCEDURE: Type RUN, press RETURN. The program responds with "WHAT FILE NUMBER?" Enter the number of the file that contains the data and press RETURN. The next prompting is "ALL STUDENTS, OR ONE STUDENT?" (ENTER ALL or ONE). Make the appropriate keyboard entry and press RETURN. If you entered ONE, an additional prompting follows: "WHICH STUDENT?" Enter the name of the student exactly as it is stored, and press RETURN.

OUTPUT SAMPLE:

```
RUN
WHAT FILE NUMBER? 1
ALL STUDENTS, OR ONE STUDENT? (ENTER "ALL" OR "ONE") ALL

STUDENT1
GRADE (1): 78
GRADE (2): 98
GRADE (3): 80
GRADE (4): 75

STUDENT2
GRADE (1): 88
GRADE (2): 95
GRADE (3): 82
GRADE (4): 80

STUDENT3
GRADE (1): 75
GRADE (2): 70
GRADE (3): 80
GRADE (4): 78

STUDENT4
GRADE (1): 90
GRADE (2): 95
GRADE (3): 85
GRADE (4): 90


RUN
WHAT FILE NUMBER? 1
ALL STUDENTS, OR ONE STUDENT? (ENTER "ALL" OR "ONE") ONE
WHICH STUDENT? STUDENT2

STUDENT2
GRADE (1): 88
GRADE (2): 95
GRADE (3): 82
GRADE (4): 80
```

Section 8

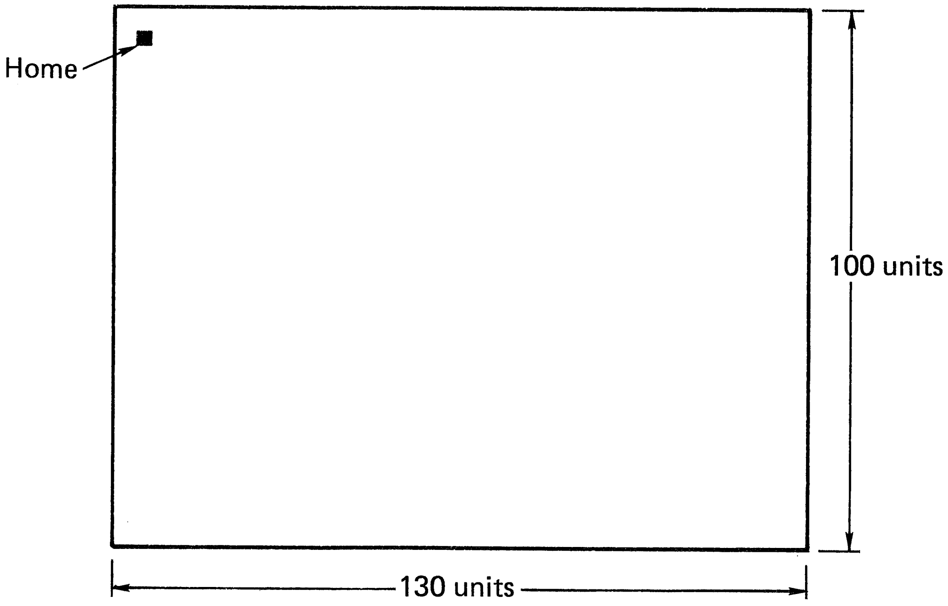
GRAPHICS

BACKGROUND

The Graphic System provides a convenient, rapid, and relatively straightforward means of using pictures (graphics) to display the results of varied and complex problems. Graphics can be used to plot and interpret data and functions, generate "computer art", produce animations, and generally promote effective communications.

The fundamental building block in graphics is a straight line. All graphical representations on the Graphic System display result from straight lines; this includes curved lines. Curves are produced by connecting a series of short, straight lines. Circles are represented similar to the way a polygon with many sides, for example, begins to look circular. The key, again, is the straight line.

Straight lines, in turn, are defined by their two end points. In order to have the Graphic System draw a line, you must specify the location of the starting point and the finishing point. These two points are specified by telling the Graphic System the horizontal and vertical coordinates of each point. More specifically, you have to move the display cursor to the point on the screen where the line is to begin, and then have the system draw a line to another point on the screen. After this operation is complete, the cursor remains at the point where the line ends. Typically, the starting point for the next line is the current position of the cursor (the ending point of the previous line).



The proportions of the screen, and the location of "home".

MOVE AND DRAW

After you have executed an INIT (for initialize) statement, the system sets the width to 130 units and the screen height to 100 units. (The term "unit" and the keyword INIT will be elaborated upon later in this section.) The screen is erased with a PAGE statement, which also sends the cursor to the "home" position at the upper left hand corner of the screen. If you want to "home" the cursor without erasing the screen, use a HOME statement.

The MOVE and DRAW Statements

In order to do graphics, you must exercise two elementary types of control over the cursor. One type of control is the ability to move the cursor into a position on the screen without drawing a visible line. The other type of control is the ability to draw a visible line between two specified points. The two statements satisfying these requirements are MOVE and DRAW. The syntax form for MOVE is

```
[Line number] MOVE numeric expression, numeric expression
```

The first numeric expression becomes the horizontal coordinate and the second numeric expression becomes the vertical coordinate. These two coordinates are frequently the X and Y coordinates, respectively. The MOVE statement allows you to position the cursor at any point on the screen without producing a visible line.

The DRAW statement has a syntax form which is essentially the same as MOVE:

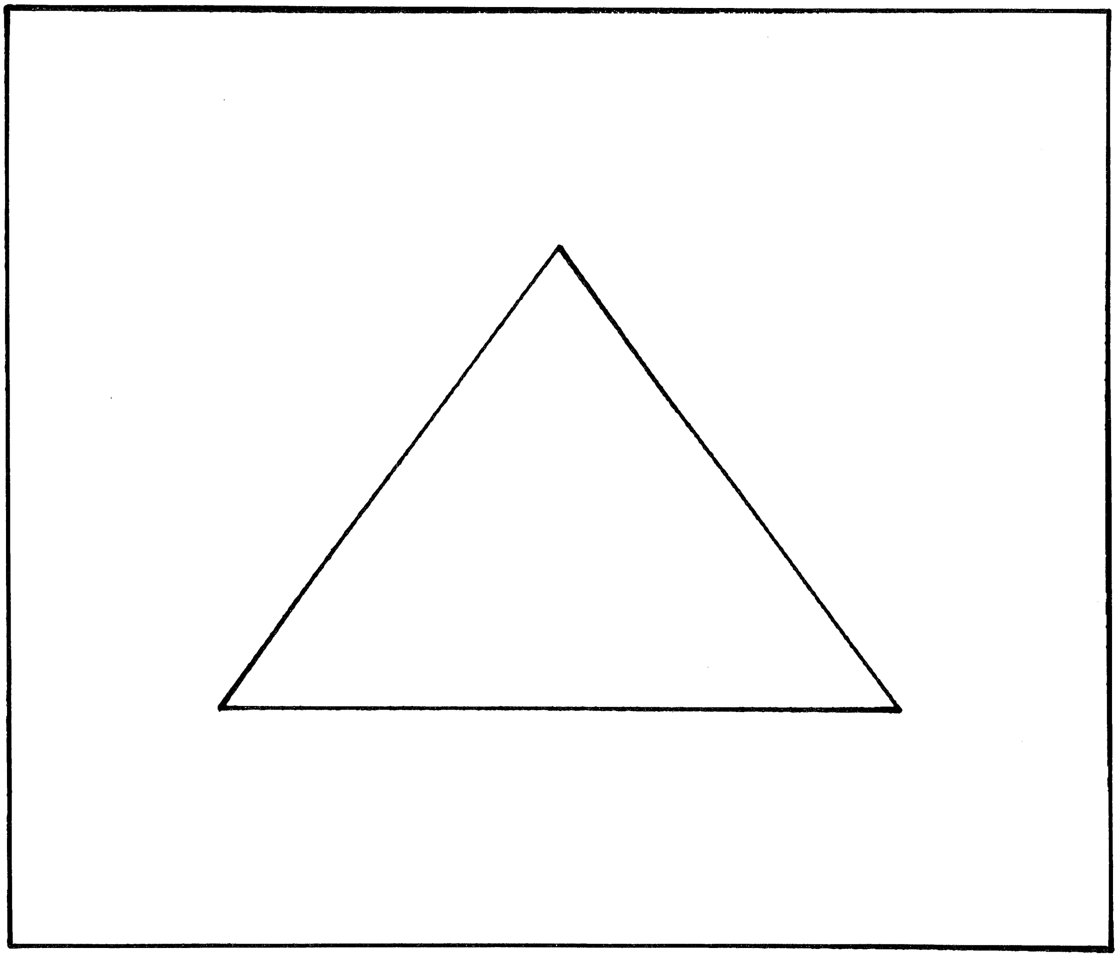
```
[Line number] DRAW numeric expression, numeric expression
```

The DRAW statement produces a visible line between the last position of the cursor and the coordinates specified by the two numeric expressions. The first numeric expression is the X coordinate and the second numeric expression is the Y coordinate.

In order to gain a feel for using MOVE and DRAW, enter and execute this program:

```
100 PAGE
110 INIT
120 MOVE 65,75
130 DRAW 25,25
140 DRAW 105,25
150 DRAW 65,75
160 END
```

When you execute this program, the PAGE statement clears the screen. Next, the INIT statement establishes environmental parameters (covered later) so that the screen space is established as 130 units horizontally by 100 units vertically. Then, statement 120 moves the cursor to the point that becomes the apex of a triangle, and from that point the triangle is drawn. The results are shown below. Notice that in MOVE and DRAW the horizontal coordinate is specified first, then the vertical. Also, positive values indicate "right" or "up"; negative values indicate "left" or "down". The conventional Cartesian coordinate system is in effect and the display screen represents the first quadrant.



MAPPING

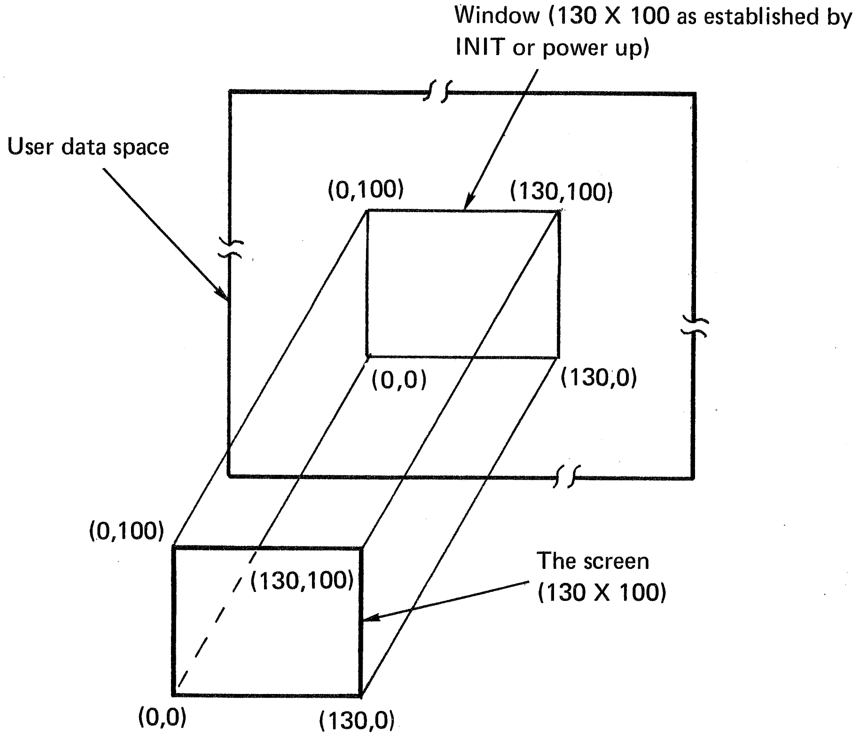
The Concept of Mapping

Now that you are acquainted with the uses of MOVE and DRAW, the next concept to be gained is called "mapping". This is an important concept. Mapping is the process the Graphic System uses to transform the data units you are working with (inches, pounds per square inch, microseconds, etc.) into the units actually used by the System to plot the data on the screen. Restated, mapping establishes the correspondence between the user data space and the screen space. Unless the user data space just happens to be a 130:100 ratio (which is the horizontal to vertical ratio on the screen), the mapping relationship between user data space and screen space has to be specified.

Establishing a Window

The largest numbers that the System can work with are −1.0E+307 and 1.0E+307. Therefore, the user data space can be regarded as a plane extending from −1.0E+307 to +1.0E+307 on both axes. Clearly, you don't always want to be presented with a plane of such proportions to work with. You need to be able to establish a "window" within this plane with a size that is appropriate to contain the information you want to graph.

After the Graphic System is powered up, or after an INIT statement is executed, the display screen window is defined as a rectangle 130 units wide and 100 units high. The horizontal axis is bounded by 0 and 130, and the vertical axis is bounded by 0 and 100. Since this corresponds to the size of the screen, this situation permits one-to-one, distortion-free mapping of the window on the user data plane to the screen space as depicted below.



In the above illustration, the user data space is still a plane extending from −1.0E+307 to +1.0E+307 in both directions. The window extends from 0 to 130 horizontally and 0 to 100 vertically. (Obviously, no attempt has been made to draw the window and user data space to scale.)

WINDOW Statement

If everybody's data fit nicely into a 130 X 100 window, the parameters established by the INIT statement (as suggested above) would be sufficient. However, this is seldom the case. To get around this potential shortcoming, the Graphic System allows you to specify a window of your own choosing. That is, you can select whatever portion of the user data space you want to see on the screen. You do this with the WINDOW statement:

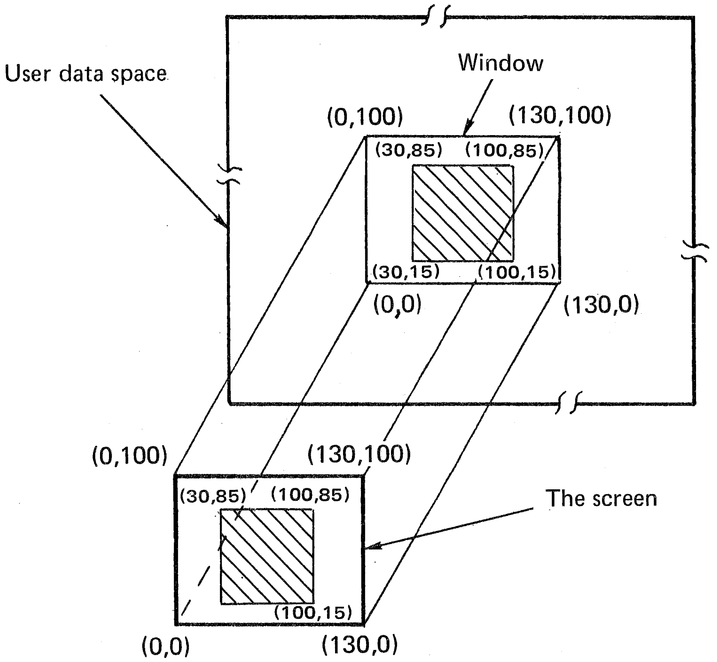
```
[Line number ] WINDOW numeric expression , numeric expression , numeric expression ,
numeric expression
```

The first numeric expression specifies the minimum horizontal value in user data unit and the second numeric expression specifies the maximum horizontal value in user data units. The third numeric expression specifies the minimum vertical value in user data units and the fourth numeric expression specifies the maximum vertical value in user data units.

One way to illustrate the effects of the WINDOW statement is by executing a program that draws a box. The program is executed several times, using different parameters each time. Here is the program:

```
100 PAGE
110 MOVE 30,15
120 DRAW 30,85
130 DRAW 100,85
140 DRAW 100,15
150 DRAW 30,15
160 END
```

Line 100 clears the screen and homes the cursor; line 110 moves the cursor to the beginning corner of the square, and the remaining lines draw the box and terminate program execution. Before running the program the first time, use an INIT statement to initialize the WINDOW parameters to their default values. These values are 0,130,0,100. Type INIT and press RETURN, then type RUN and press RETURN. The following illustration shows the relationship between the plotted square, the window, and the screen. (The square is shaded for clarity.)

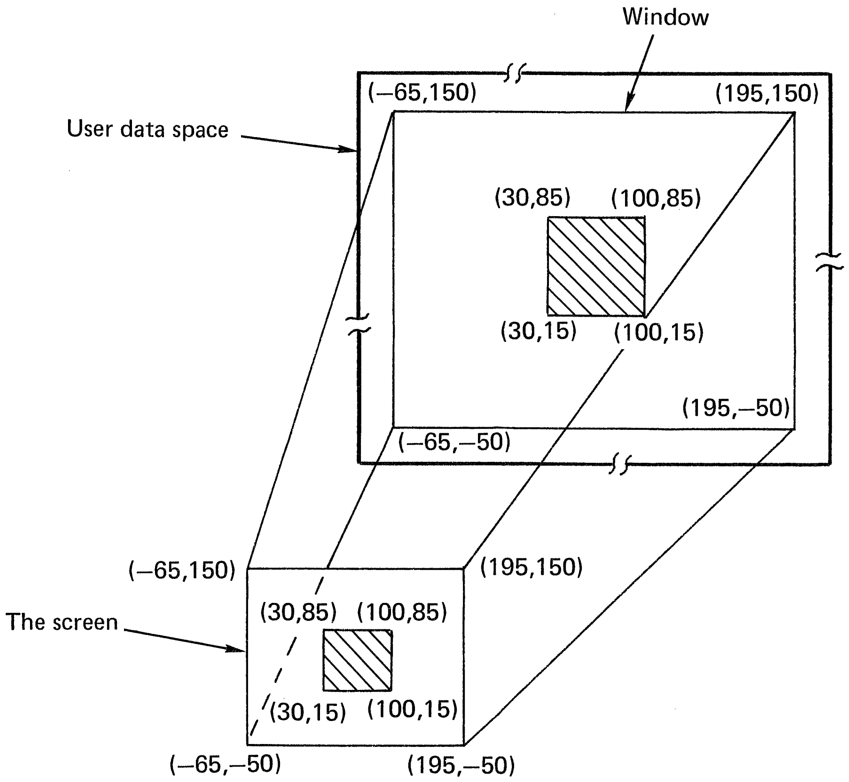


Now run the program again but this time double the size of the window. This is done by entering the following statement

```
WINDOW –65,195,–50,150
```

and pressing RETURN. (These parameters were chosen because they double each edge of the window and still keep the square centered.) Run the program again observe the square that is produced.

Notice this time that the square appears to be half as large as the one produced earlier. The reason for this is that the window (or portion of the user data space being observed) is twice as large. The square itself, however, is plotted with the same coordinates as before. Compare the following diagram with the previous one.

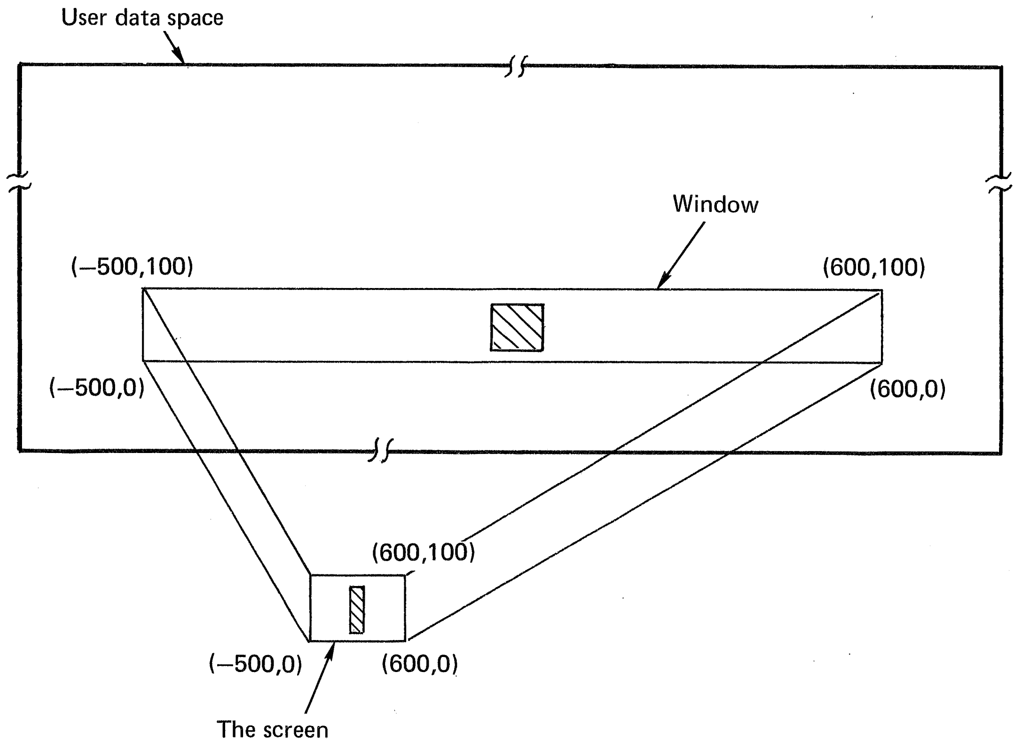


In effect, twice as much user data space (actually, four times the area) is represented on the screen with a corresponding reduction in the appearance of the square with this fixed area. It is as though you have a variable lens and have zoomed out to look at a greater area.

The horizontal and vertical parameters of the WINDOW statement can be changed independently. Try running the program again, this time using

```
WINDOW –500,600,0,100
```

This establishes a window that is 1110 user data units wide, and 100 user data units high. The vertical size is the same as the default values, but the horizontal part is nearly 10 times greater than the default setting of 130. As a result, the box produced is quite narrow in appearance as suggested below.



Graphic Display Units

The earlier discussion about the WINDOW statement involved the user data space, and the unit of measurement can be most anything — microvolts, pounds, miles, furlongs, etc. The unit of measurement used internally for graphics, however, is always the same. This unit is called the Graphic Display Unit, or GDU. A GDU is one hundredth of the vertical dimension of the screen. The screen is precisely defined internally as being 100 GDUs high and 130 GDUs wide. A horizontal line drawn at the midpoint of the screen is therefore 50 GDUs above the bottom edge, and two horizontal lines that trisect the screen are 33.3 and 66.7 GDUs from the bottom of the screen (approximately).

The only time you need to worry about GDUs is when you establish the drawing boundaries on the screen with the VIEWPORT statement and the scale factor with the SCALE statement. The rest of the graphic statements allow you to work in your own units of measure, whatever they are.

Establishing a Viewport

The "window" concept, discussed on preceding pages, deals with the problem of specifying the portion of the user data space that is represented on the screen. "Viewport", on the other hand, refers to the drawing boundaries on the display screen; that is, the area containing the information in the window. The desired portion of the screen is specified in terms of GDUs.

The VIEWPORT statement has the following syntax form:

```
[Line number] VIEWPORT numeric expression , numeric expression , numeric expression ,
numeric expression
```

As with WINDOW, the four parameters refer to horizontal and vertical bounds, and all four can be numeric expressions. The units of measure, however, are GDU's. The first parameter is the horizontal minimum, the second parameter the horizontal maximum, the third parameter the vertical minimum and the fourth parameter the vertical maximum.

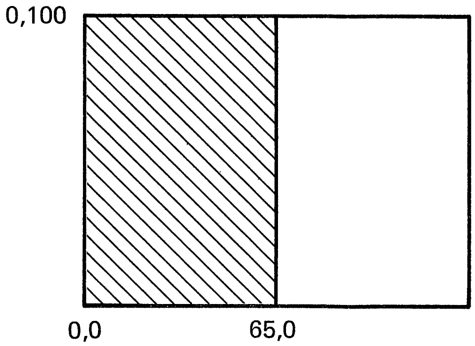
Since the screen is 130 GDU's wide and 100 GDU's high, the statement that allows a plot to fill the entire screen is one like

```
VIEWPORT 0,130,0,100
```

If, for some reason, you want to restrict your graph to the left half of the screen, you can use something like

```
VIEWPORT 0,65,0,100
```

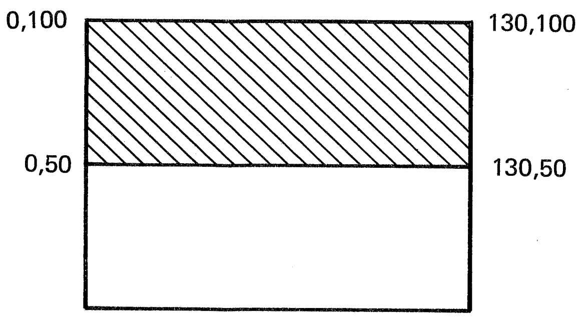
This enables you to use the area of the screen represented by the shaded area below:



Conversely, if you wanted to use only the upper half of the screen, you can specify

VIEWPORT 0,130,50,100

This drawing area is represented by the shaded area in the following illustration:



The INIT statement, which initialized the WINDOW parameters to 0,130,0,100, also initializes the VIEWPORT parameters. Not surprisingly, the initialized parameters for VIEWPORT parallel those for WINDOW: they are 0,130,0,100. This means that in the square box program discussed earlier, you were also using VIEWPORT but didn't know it. The INIT statement established the 0,130,0,100 values for both WINDOW and VIEWPORT. (Turning the power on also establishes these parameters to their default values.)

The following explanation of VIEWPORT utilizes the square box program that was used to support the discussion about WINDOW. Here is the program again:

```
100 PAGE
110 MOVE 30,15
120 DRAW 30,85
130 DRAW 100,85
140 DRAW 100,15
150 DRAW 30,15
160 END
```

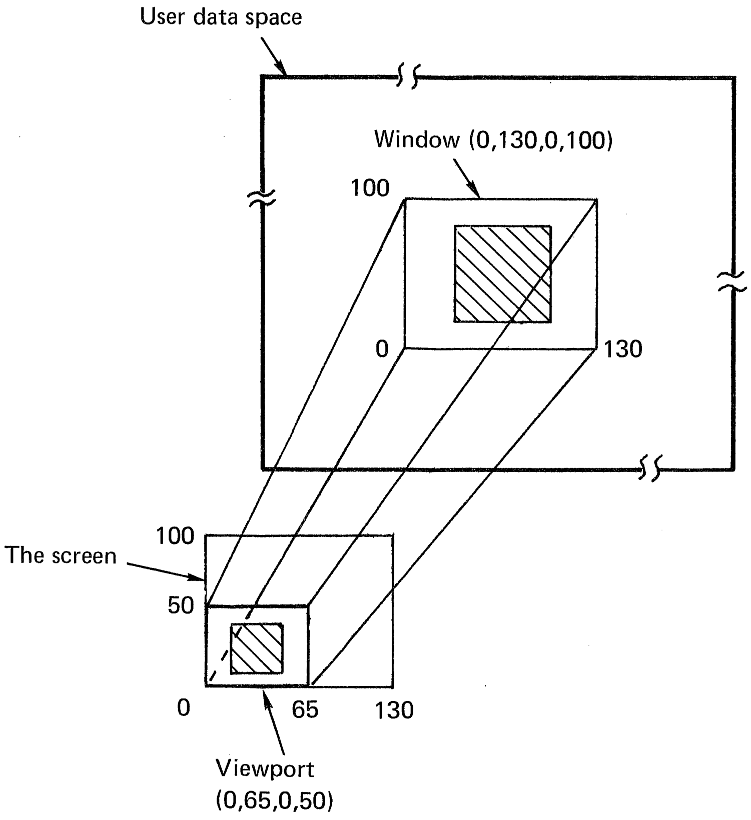
Verify that the program is located in memory by entering

```
INIT
RUN
```

The large square box should be centered on the screen. Now try the following:

```
VIEWPORT 0,65,0,50
RUN
```

This causes the square box to again appear, but this time it is located in the lower left corner of the screen. The reason for this is that the VIEWPORT statement defines the area on the screen bounded by 0 and 65 GDU's horizontally and 0 through 50 GDU's vertically as the drawing surface. Refer to the following diagram:

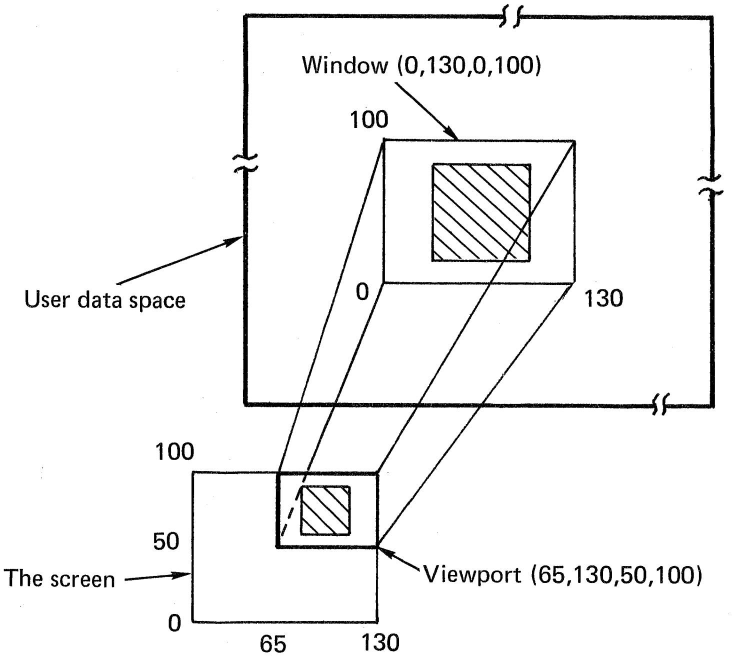


Up to this point, the window has been mapped onto the entire screen space. Now, the window is mapped onto a segment of the screen, and the only change necessary is to alter the VIEWPORT parameters.

Run the program again, but this time make the plot appear on the upper right quadrant of the screen.

```
VIEWPORT 65,130,50,100
RUN
```

The results are shown below.



Here again, the contents of the window are mapped onto the area on the screen specified by VIEWPORT. A change in the VIEWPORT parameters is the only thing necessary to relocate the drawing area.

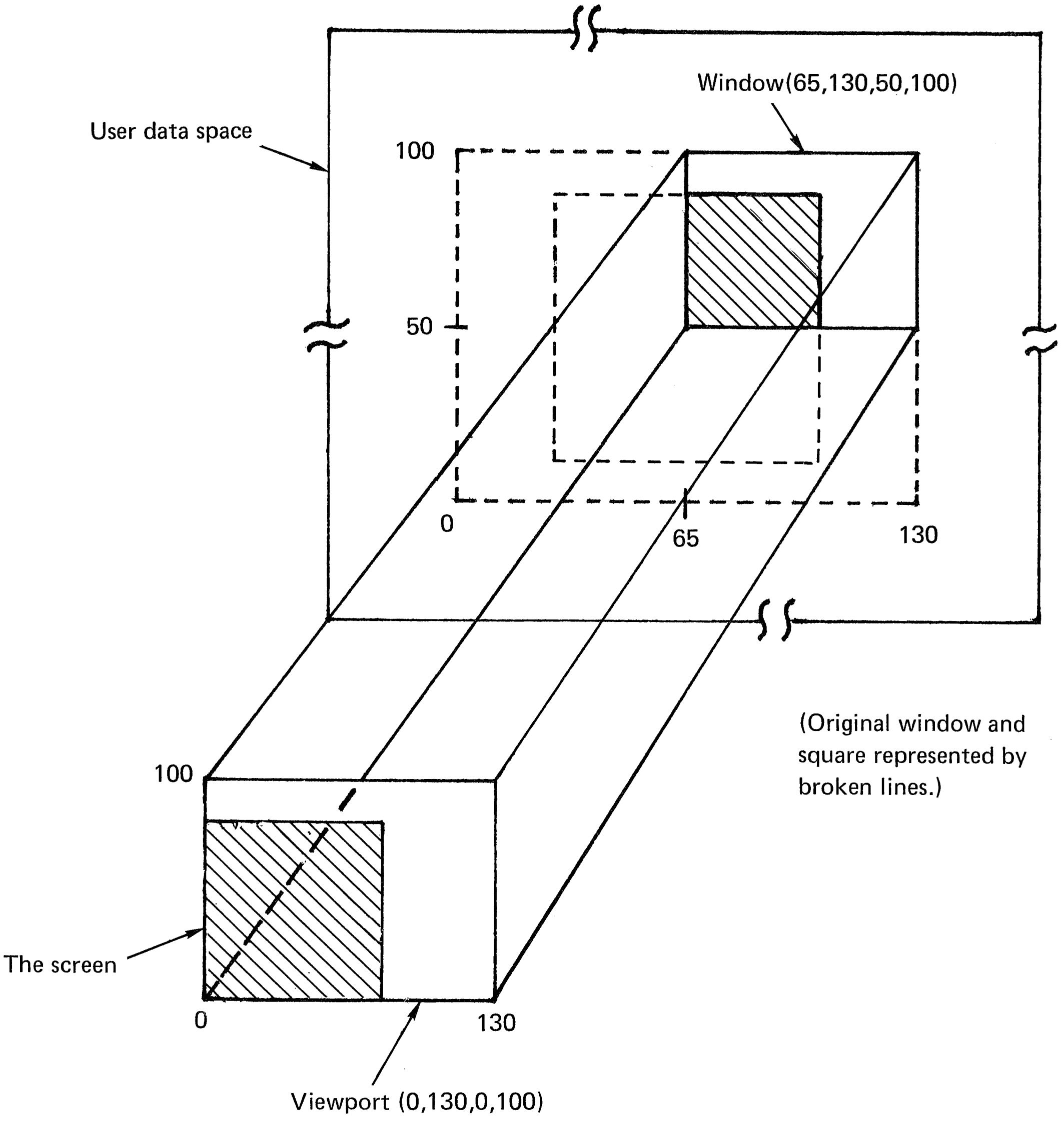
WINDOW and VIEWPORT Combined

To summarize the preceding pages, WINDOW allows you to select the portion of the user data space you want to see, and VIEWPORT allows you to select the area of the screen that the window is mapped into. The parameters associated with WINDOW are expressed in user data units. The parameters of VIEWPORT are expressed in GDU's.

To illustrate the effects of WINDOW and VIEWPORT acting in combination, run the square box program again. Use the default parameters for VIEWPORT, and specify the parameters for WINDOW.

```
INIT
WINDOW 65,130,50,100
RUN
```

The coordinates of the square box are based on an original window of 0,130,0,100. The new parameters for WINDOW restrict the plot to the right corner of the box. The VIEWPORT parameters are unchanged, so the entire screen is utilized. The results are shown below.

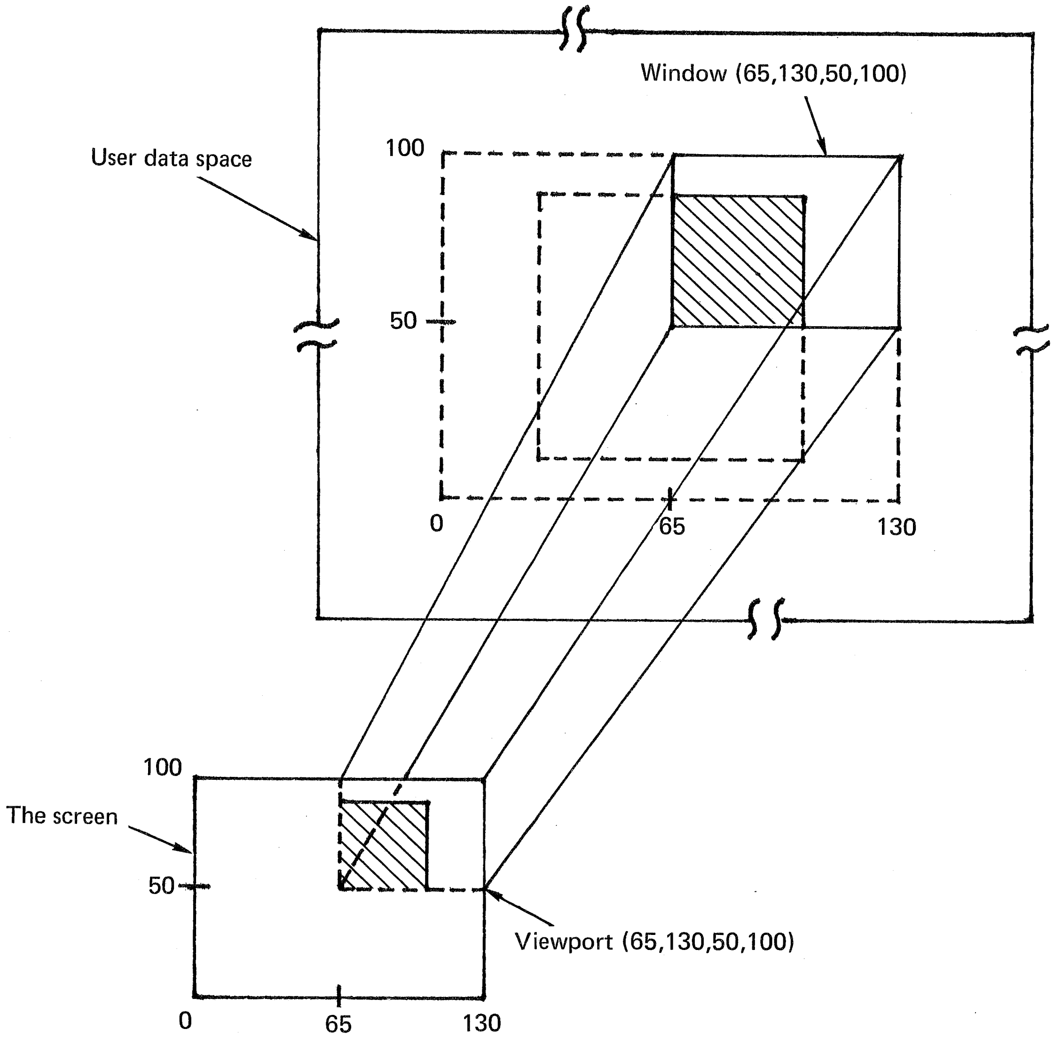


(Original window and square represented by broken lines.)

Run the square box program again, only this time change the VIEWPORT parameters:

```
VIEWPORT 65,130,50,100
RUN
```

The WINDOW parameters are still 65,130,50,100 from the previous illustration. The result of this latest run (below) is that the same information is plotted into a smaller area specified by the new VIEWPORT parameters. The upper right corner of the square appears in the upper right corner of the screen, and nothing else appears. This occurs because the WINDOW statement specifies what it is you are going to see, and graphic information outside the WINDOW is not shown. This effect is called *clipping*.



The SCALE Statement

The WINDOW statement shown in preceding pages defines the portion of the user data space that is represented on the screen. The companion of the WINDOW statement is the SCALE statement. The SCALE statement provides an alternate method of establishing a window. The syntax form is

```
[Line number] SCALE numeric expression, numeric expression
```

The first numeric expression determines the horizontal scale factor and the second numeric expression determines the vertical scale factor.

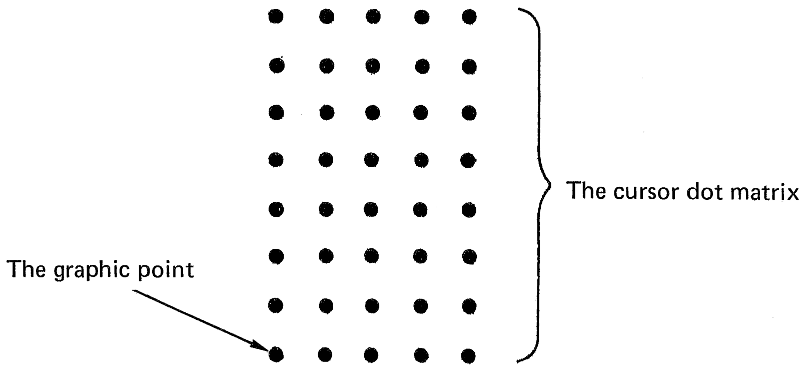
The term "scale factor" refers to the ratio of user data units per GDU. For example, the statement

```
150 SCALE 100,100
```

means that each GDU is equivalent to 100 user data units. The default parameters for SCALE are 1 and 1. These parameters are set on power up and after the execution of an INIT statement. These particular default values are used because both WINDOW and VIEWPORT have default values of 0,130,0,100. This says that one GDU is equivalent to one user data unit.

The WINDOW statement explicity defines the limits of the user data space that is mapped into the VIEWPORT. This implies certain scale factors involving user data units per GDU. For example, if the WINDOW is specified as 0,260,0,200 and the VIEWPORT is 0,130,0,100 then the implied scale factors are 2 user data units per GDU horizontally and vertically (or SCALE 2,2). This touches upon the principal difference between WINDOW and SCALE. The SCALE statement explicitly defines the scale factors and this, in conjunction with the selected VIEWPORT, implicitly defines the limits of the user data space that is viewed. For example, if you specify SCALE 2,2 and VIEWPORT 0,130,0,100, then the implied window is 260 units wide and 200 units high on the user data space.

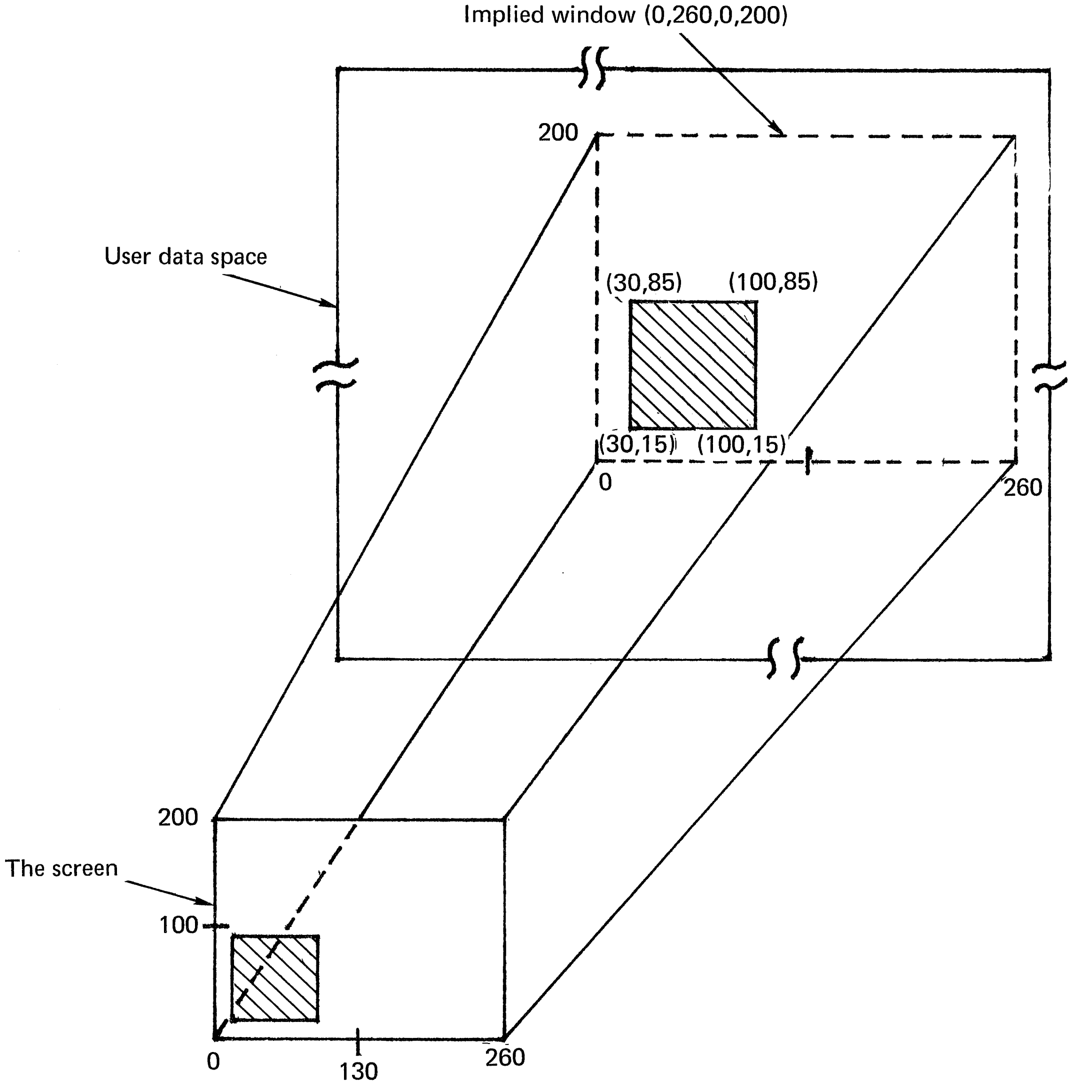
SCALE carries with it an additional feature in that the location of the point of origin (0,0) (on the user data space) is defined as the location of the graphic point when SCALE is executed. The graphic point is the lower left dot on the 5 X 8 dot matrix of the alphanumeric cursor. This means that if the cursor is located in the lower left corner of the screen when SCALE is executed, then (0,0) is also in the lower left corner of the screen. It also means that if the cursor is at the top of the screen when SCALE is executed, the point of origin is established at the top of the screen. If the plot happens to contain all positive values then, in this case, the "plot" is located somewhere above the screen (i.e., it will be totally clipped). The point is this: when using SCALE, the position of the cursor is important because it establishes the point of origin.



Run the square box program again, this time using SCALE. Keeping in mind the fact that SCALE regards (0,0) as the location of the cursor (or graphic point) when the SCALE is executed. Precede the SCALE with a MOVE as follows:

```
 INIT
MOVE 0,0
SCALE 2,2
RUN
```

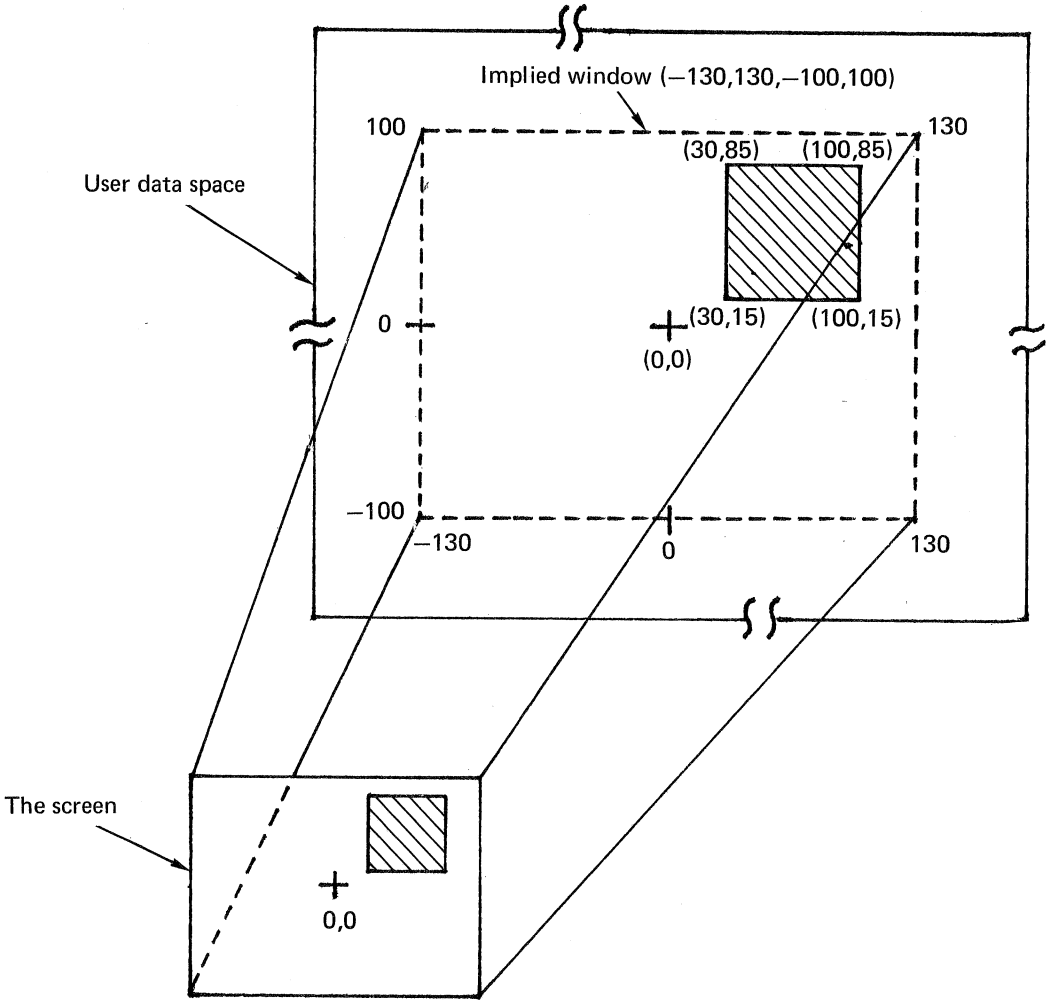
The box appears half size in the lower left area of the screen. It is half size because SCALE causes each GDU to represent 2 user data units, and the box was originally drawn with each GDU being equivalent to 1 user data unit. The screen is now showing twice as much user data space along each axis, or four times the area. SCALE 2,2 has produced a "zoom out" effect with respect to the user data space. This is represented in the following diagram.



The familiar square box can be easily shifted to the upper right quadrant of the screen by taking advantage of the fact that SCALE establishes the point of origin as the location of the cursor at the time SCALE is executed:

```
INIT
MOVE 65,50
SCALE 2,2
RUN
```

Since the MOVE places the cursor at mid-screen prior to execution of SCALE, that location is transformed from (65,50) to (0,0) by the SCALE statement. Mid-screen is now (0,0) in terms of user data units; it is no longer (65,50). The following diagram illustrates the situation after the program is run.



You have probably noticed the frequent use of the INIT statement in the previous examples. INIT, as suggested earlier, establishes a common starting point. The INIT statement sets up the following default values:

```
WINDOW 0,130,0,100
VIEWPORT 0,130,0,100
SCALE 1,1
```

When you use INIT, it becomes easy to move the cursor to, say, mid-screen, because the user data units and the GDU's are conceptually the same. You just say MOVE 65,50. Similarly, if you want to move the cursor to the lower left corner, this can be done with MOVE 0,0. The lower right is at (130,0), etc. Without the INIT, you may not know for sure where you are; (0,0) could be off screen or anywhere.

GRAPHIC OUTPUT

The RMOVE and RDRAW Statements

At the beginning of this section, MOVE and DRAW were discussed. MOVE was shown to cause the cursor, or graphic point, to be positioned at the specified location without producing a visible line. DRAW causes a visible line to be generated. Both MOVE and DRAW deal with "absolute" locations; that is, they cause the cursor to move from its present location to some other location on the screen which is specified in terms of user data units. Absolute locations are defined by horizontal and vertical coordinates which correspond to an interval along each axis.

"Relative" locations differ considerably from "absolute" locations. Relative locations are defined by a horizontal and vertical interval away from the present location of the cursor instead of an interval along each axis. The following two statements permit you to move the cursor to a location which is relative to its present location:

```
[Line number] RMOVE numeric expression, numeric expression
```

and

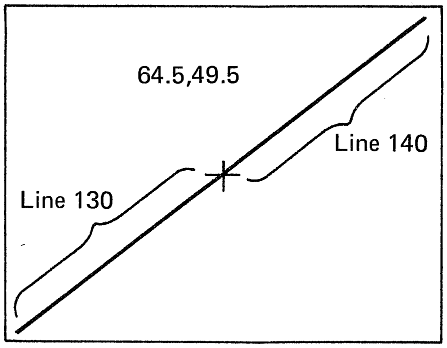
```
[Line number] RDRAW numeric expression, numeric expression
```

In each case, the first numeric expression and the second numeric expression refer to the relative horizontal and vertical distances, respectively. RMOVE is similar to MOVE in that a visible line does not result from its use. RDRAW is similar to DRAW in that is produces a visible line. The "R" in RDRAW and RMOVE stands for "relative".

The program segment below uses a DRAW and an RDRAW statement with the same parameters:

```
100 PAGE
110 INIT
120 MOVE 0,0
130 DRAW 64.5,49.5
140 RDRAW 64.5,49.5
```

Line 130 causes a diagonal line to be drawn from the lower left corner of the screen to a point near mid-screen. (These particular parameters were chosen because they show that non-integer values can be specified.) Line 140 extends the diagonal line to a point very near the upper right corner; this draws a line to a point which is 64.5 units to the right and 49.5 units above the ending point of the previous line.



The ROTATE Statement

Clearly, a useful capability for graphics is the means to rotate a given line through an arc. This capability can be generated with a complex combination of relative moves and draws involving various trigonometric techniques; however, it is very impractical. A much easier approach is to use the ROTATE statement.

```
[Line number] ROTATE numeric expression
```

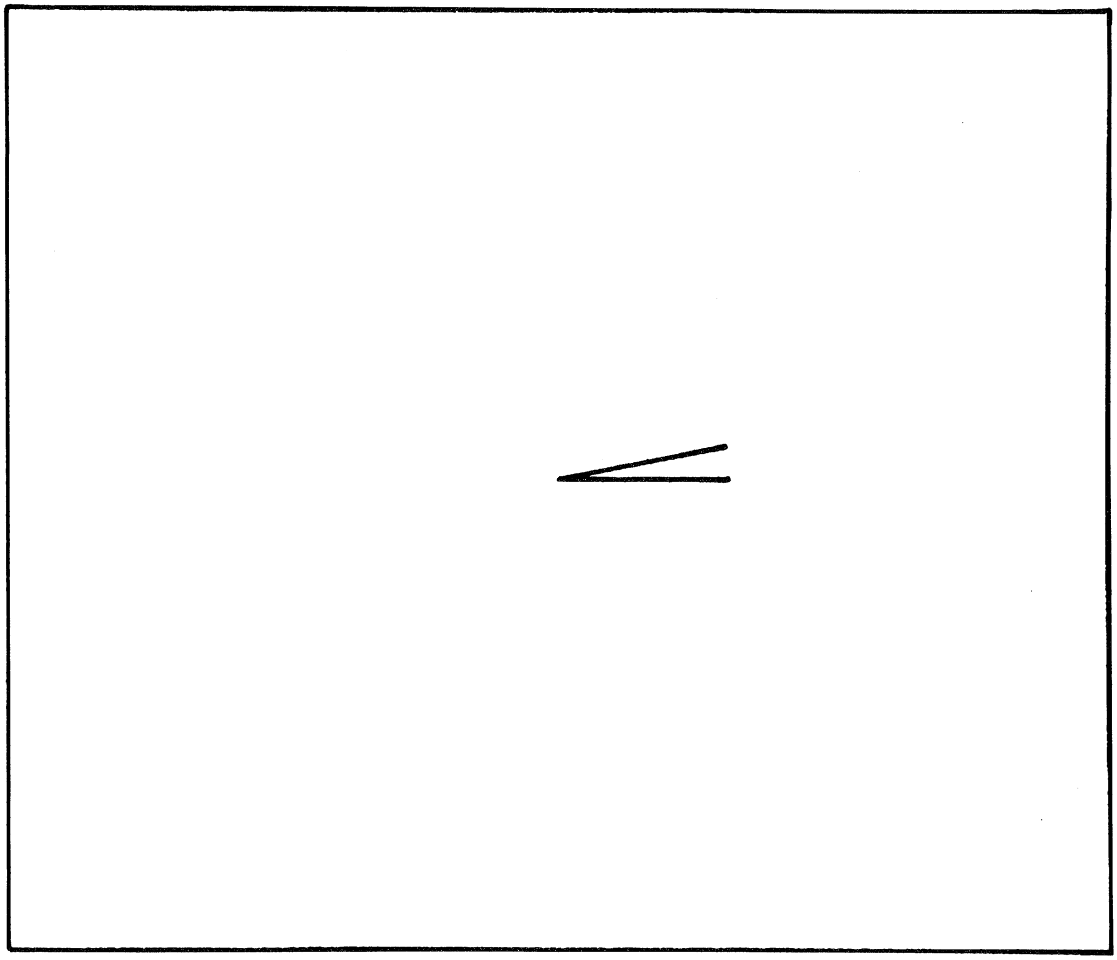
The *numeric expression* indicates the number of degrees, radians, or grads through which a specified vector (line) rotates. Positive values yield counterclockwise rotation; negative values yield clockwise rotation. This particular statement involves a certain amount of programming overhead; i.e., there are conditions which have to be established prior to using ROTATE. Among these conditions are the specification of the appropriate trigonometric units (degrees, radians or grads) as suggested above. To rotate a line 45 degrees, you must execute a SET DEGREE statement before you execute the ROTATE statement. In addition, ROTATE works only with RMOVE and RDRAW. This makes sense when you consider the apparent contradiction of attempting to rotate a line that is defined in terms of absolute coordinates. Again, rotation only applies to relative moves and draws. Example:

```
100 PAGE
110 INIT
120 MOVE 65,50
130 RDRAW 20,0
```

The above statements place a horizontal line 20 data units long with its beginning point centered at mid-screen. Now, to rotate this line 10 degrees counterclockwise:

```
140 SET DEGREES
150 ROTATE 10
160 MOVE 65,50
170 RDRAW 20,0
```

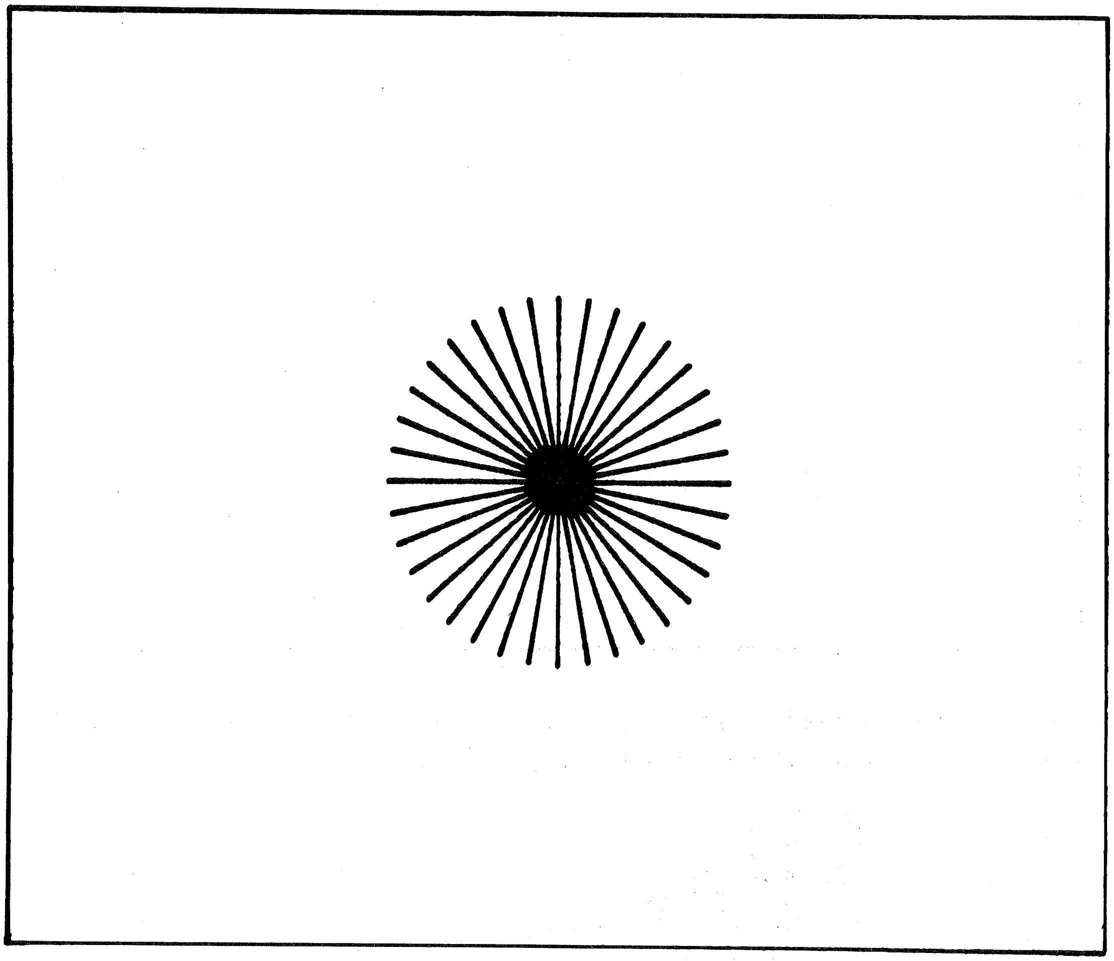
After typing RUN, the screen contains 2 lines, each 20 data units long, and with 10 degrees of arc separating them. Notice that the rotation occurs at the beginning point of each line. The MOVE is included because it places the graphic point back to mid-screen. Another way to regard the situation is that the rotation takes place at the ending point of the last absolute MOVE. (Notice that the MOVE is not affected by the ROTATE statement.)



As a further example, the following program generates a fan of 36 straight lines, each line 20 data units long, and each line rotated 10 degrees from the previous one.

```
100 PAGE
110 INIT
120 MOVE 65,50
130 SET DEGREES
140 FOR I=0 TO 360 STEP 10
150 RDRAW 20,0
160 MOVE 65,50
170 ROTATE I
180 NEXT I
190 HOME
200 END
```

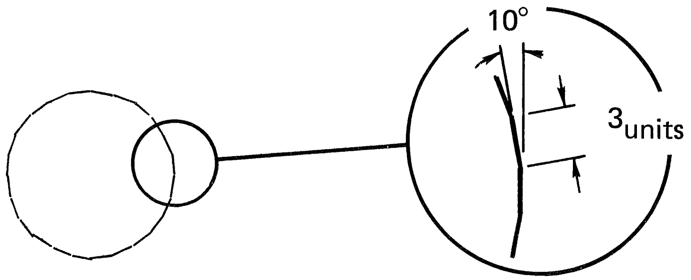
Here again, the rotation is around the beginning point of each line, and around the end point of the absolute MOVE. Essentially, this program is the same as the previous program that produced the two lines. The principal difference is the addition of the loop which increments the rotation angle. The only potentially disconcerting thing is in line 150 (the RDRAW statement). The two RDRAW parameters tell the system to draw a line 20 units away horizontally and 0 units away vertically relative to the present location of the graphic point. This is straightforward with respect to the first line that is drawn, but it begins to seem a little strange by the time the ninth line is generated. The "horizontal" parameter is now vertical. The reason for this is that the ROTATE statement produces an angular displacement in the orientation of relative moves and draws. The angular displacement is relative to the "normal" orientation (the orientation that results from the absence of ROTATE). The display which results is similar to the following:



The fan can be changed to a circle by only minor changes to the program:

```
100 PAGE
110 INIT
120 MOVE 65,50
130 SET DEGREES
140 FOR I=0 TO 360 STEP 10
150 ROTATE I
160 RDRAW 0,3
170 NEXT I
180 HOME
190 END
```

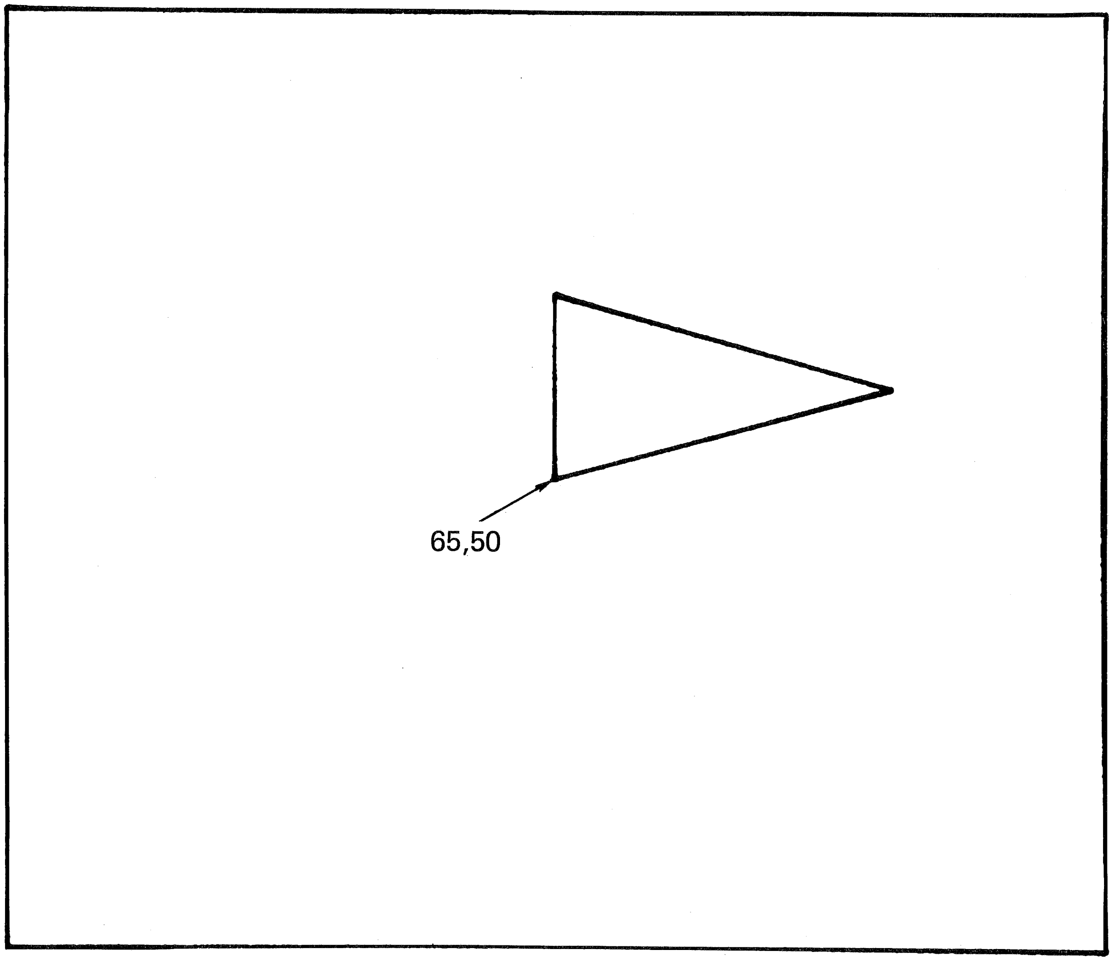
The main difference between the two programs is that the graphic point is not moved back to mid-screen after each RDRAW. Also, the first line drawn is vertical, not horizontal. The resulting plot is a circle composed of 36 straight lines, each line 3 units in length and offset from adjoining lines by 10 degrees. Notice that each RDRAW is rotated with respect to "normal" orientation, and not with respect to the previous RDRAW. A stylized version of the output is given below:



An important relationship to be aware of is that a ROTATE preceding a series of contiguous relative moves and draws causes the entire resultant plot to be rotated. Consider the following program:

```
100 PAGE
110 INIT
120 MOVE 65,50
130 RDRAW 40,10
140 RDRAW -40,10
150 RDRAW 0,-20
160 END
```

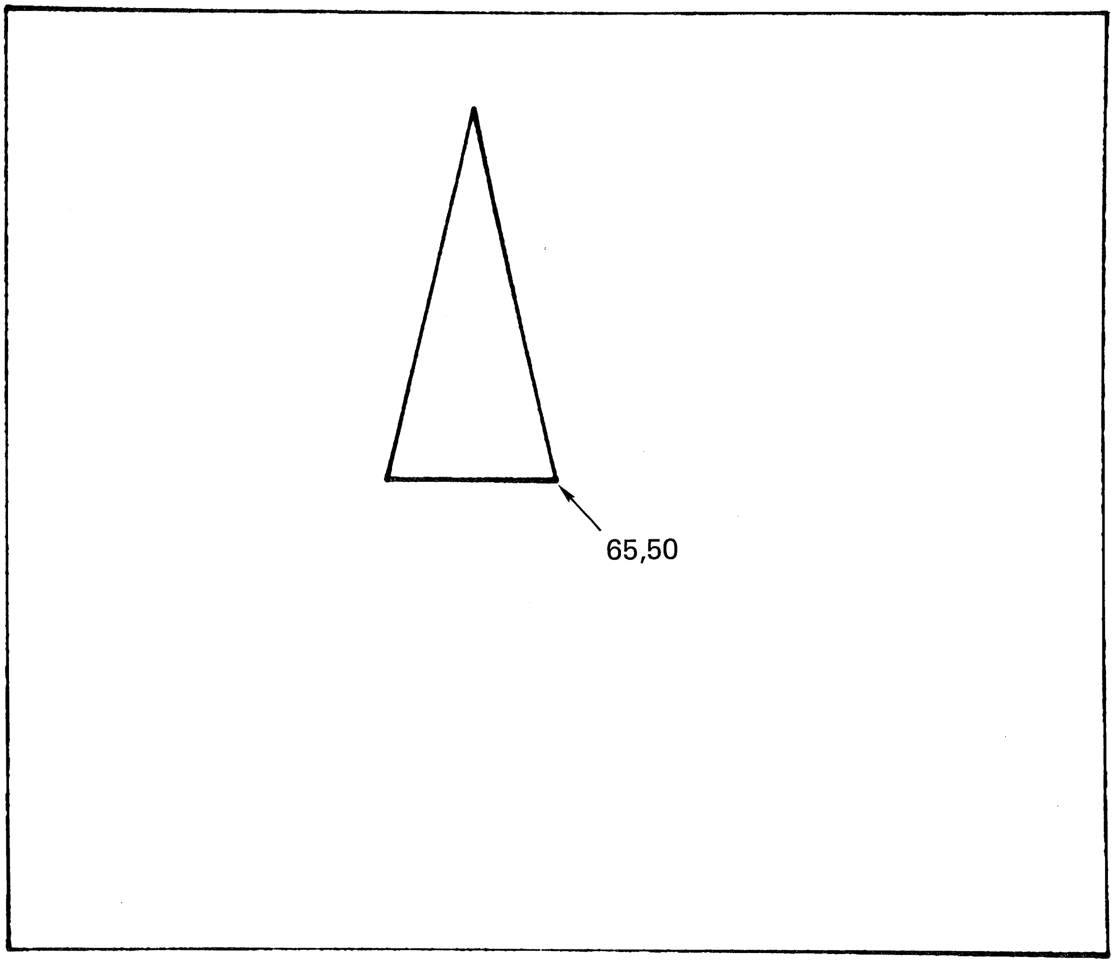
When you run this program it results in the generation of a triangle.



Now, run the program again, but first add two lines

```
121 SET DEGREE
122 ROTATE 90
```

The plot now looks like this:



As you can see, the entire plot is now rotated 90 degrees counterclockwise. The ROTATE 90 that you entered into memory caused the result of the three RDRAW statements to be rotated. The rotation point is around the end point of the absolute MOVE statement (65,50).

Using Two Arrays to Draw a Graph

A valuable aspect of graphics lies in its ability to plot data, and the most convenient way to store data is to place the data into an array. Data can be stored in arrays through a variety of methods: FOR/NEXT LOOPS, INPUT, READ, etc. Quite often, data to be plotted consists of X and Y (or horizontal and vertical) coordinate pairs. With this in mind, a good way to organize such data is to place it in two spearate arrays, such as an array "X" (for horizontal coordinates) and an array "Y" (for vertical coordinates). This way, the coordinate pairs can be readily assembled into data point. Point (1) consists of the coordinates X(1), Y(1); point (2) in this scheme consists of X(2), Y(2) and point (n) is X(n), Y(n).

This approach is facilitated in this version of BASIC by the fact that DRAW, RDRAW, MOVE, and RMOVE allow arrays to be specified as parameters.

This means that you can, for instance, store horizontal data in X, vertical data in Y, and obtain the entire graph by executing the statement

```
DRAW X,Y
```

If you have N data pairs, this is equivalent to saying:

```
FOR I = 1 to N
DRAW X(I), Y(I)
NEXT I
```

As you can see, this results in less programming effort.

Notice the difference between the manner in which a DRAW and PRINT operate when specifying arrays. If you execute

```
PRINT X,Y
```

then the display contains numeric values corresponding to X(1), X(2), X(3) . . . .X(n), Y(1), Y(2), Y(3). . . .Y(n). The PRINT statement outputs all elements in array X, and then outputs the elements in array Y. This is quite different from the statement

```
DRAW X,Y
```

which outputs the first element in X, then the first element in Y and so on until the contents of one or both arrays are exhausted.

Drawing Axis Lines

Graphs become much more meaningful when they include axis lines which provide reference points for the particular coordinate system you are using. A further refinement to the idea of axis lines is the inclusion of "tic" marks along each axis to identify the data intervals.

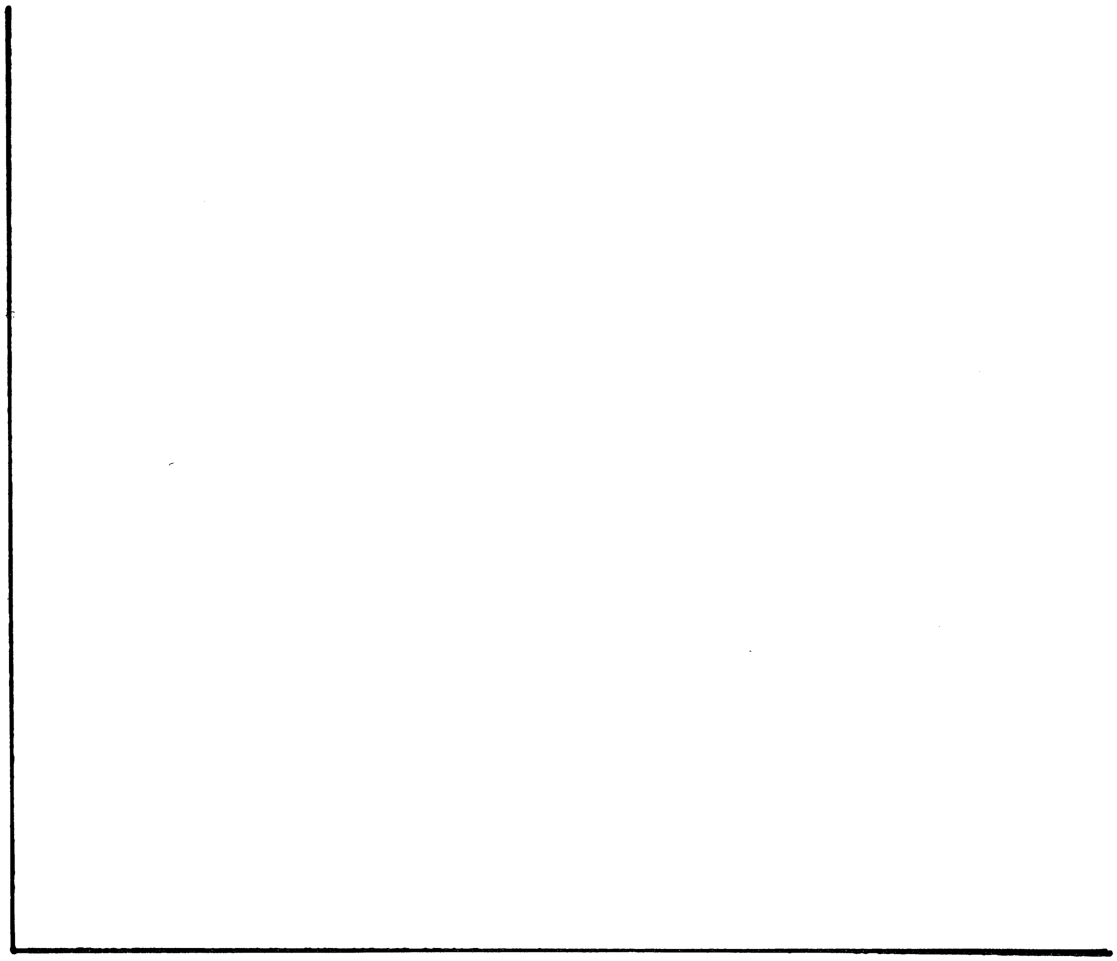
Axis lines and tic marks can of course be produced by using subroutines containing a number of DRAWs, MOVEs, etc., but that is a lot of work. All that effort is avoided with the AXIS statement. This AXIS statement is a four-parameter statement which can be used without parameters, with two parameters, or with all four parameters. The complete syntax form for the statement is given later in this discussion, which allows time to lead up to the four-parameter form.

The AXIS Statement Without Parameters

The AXIS statement used by itself (without parameters) draws a horizontal and vertical line through the point of origin in the user data space. The origin is the point at which X = 0 and Y = 0. If this point is not inside the current window location, the axis lines are drawn through the minimum X and Y data values. This is located at the left edge of the window for the vertical (Y) axis, and the bottom edge for the horizontal (X) axis. For example:

```
100 PAGE
110 INIT
120 AXIS
130 END
```

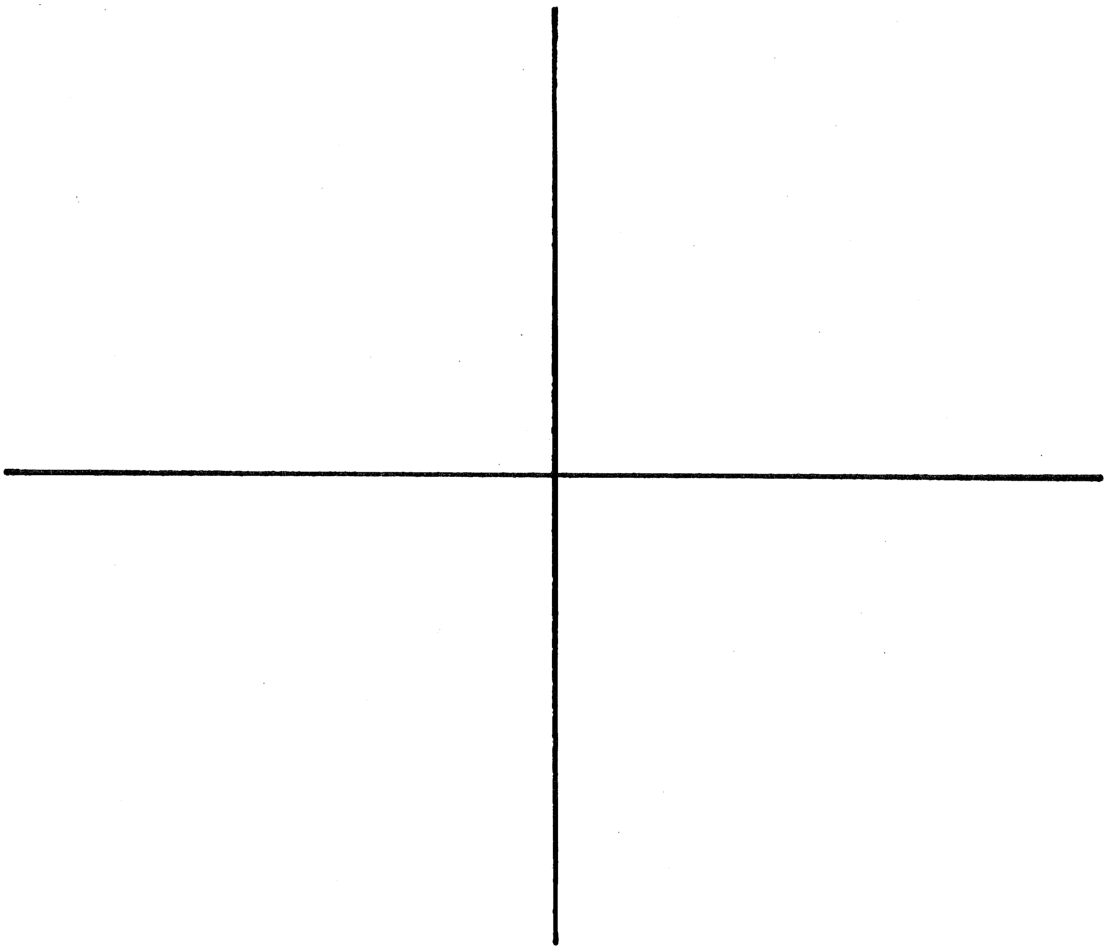
If you run the above program, the axis lines are drawn as shown below:



The axes are drawn as shown because the INIT statement is equivalent to WINDOW 0,130,0,100 and the point of origin (where X = 0 and Y = 0) is at the lower left corner of the screen. Another example:

```
100 PAGE
110 INIT
120 WINDOW -50,50,-100,100
130 AXIS
140 END
```

When you run this program, the axes produced look like this:



The axis lines (above) cross at midscreen because that is where the origin is located as established by the WINDOW statement in line 120. Another example:

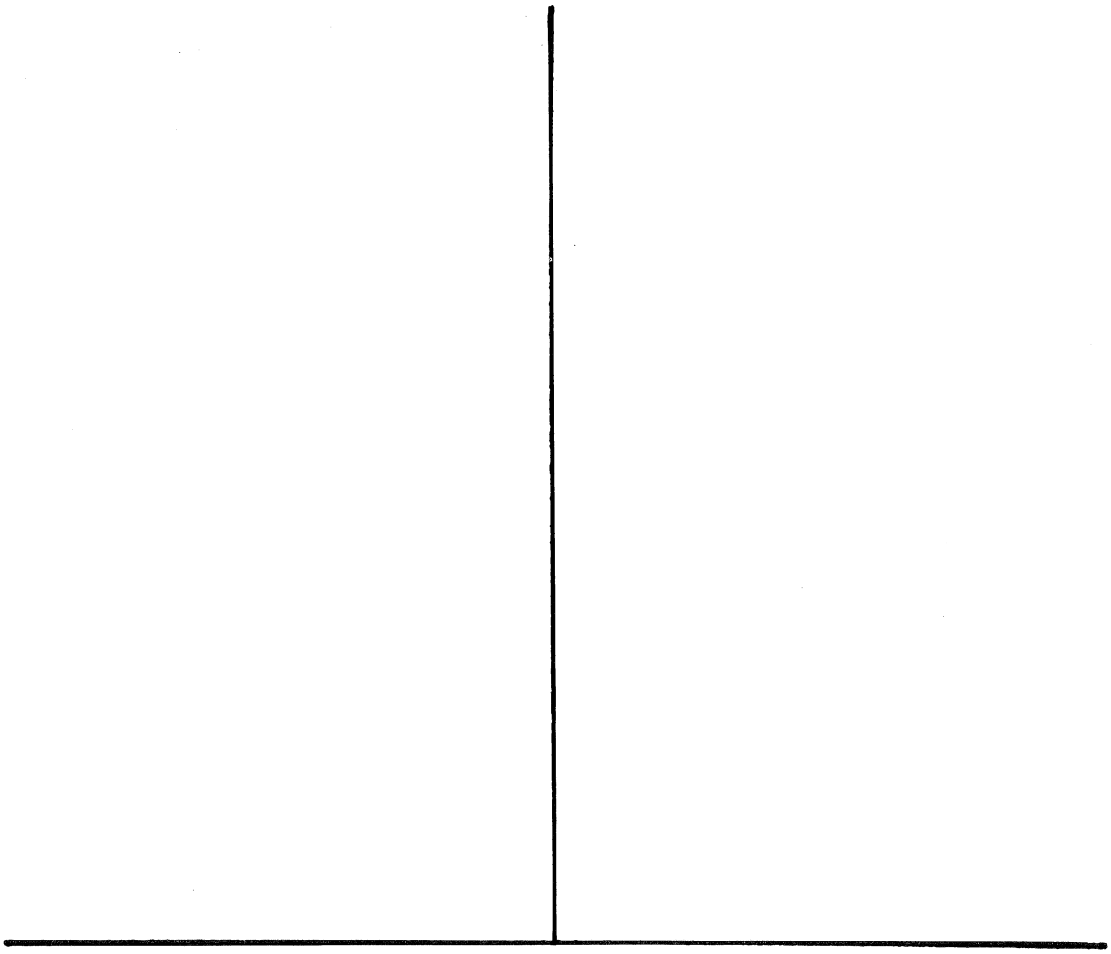
```
100 PAGE
110 INIT
120 WINDOW -100,-50,-100,-50
130 AXIS
140 END
```

This program produces a WINDOW such that the point of origin is outside the WINDOW. Therefore, the axis lines intersect in the lower left corner of the screen where minimum values for X and Y occur. See the illustration below:

In the following example, the point where X = 0 is at the middle of the horizontal axis, but the point where Y = 0 is outside the window.

```
100 PAGE
110 INIT
120 WINDOW -50,50,50,100
130 AXIS
140 END
```

The vertical axis is at midscreen (where X = 0), and the horizontal axis is at the bottom of the screen (where the minimum Y value is located).



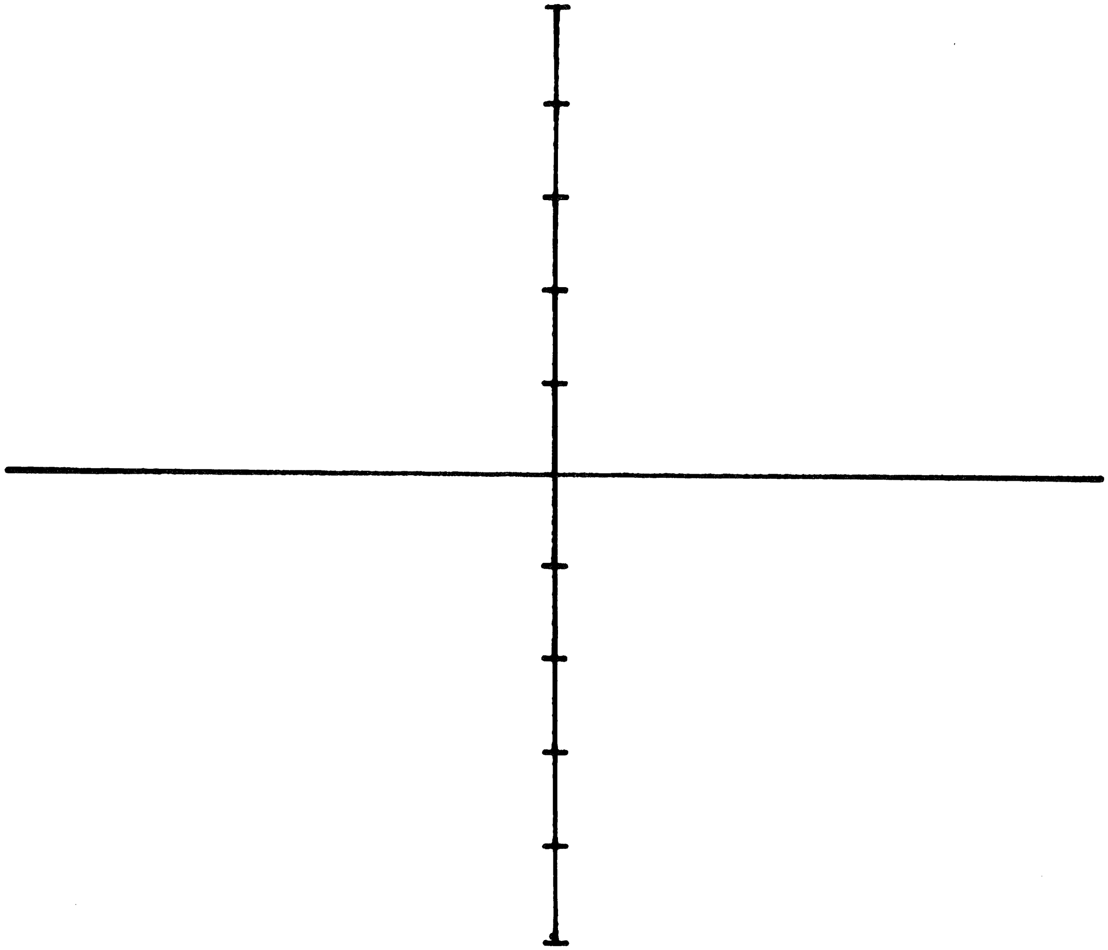
The AXIS Statement With Two Parameters

Up to this point, the discussion about axis lines has centered around the AXIS statement without parameters. At the beginning of the discussion, it was suggested that the AXIS statement can be used to produce tic marks.

The interval between tic marks, specified in user data units, is determined by the first two parameters of the AXIS statement. The first parameter is the interval between tic marks on the horizontal (X) axis; the second parameter is the interval between tic marks on the vertical (Y) axis. If either parameter is specified as zero, the corresponding axes are drawn with no tic marks. For example:

```
100 PAGE
110 INIT
120 WINDOW -50,50,-50,50
130 AXIS 0,10
140 END
```

This generates an output like the following:

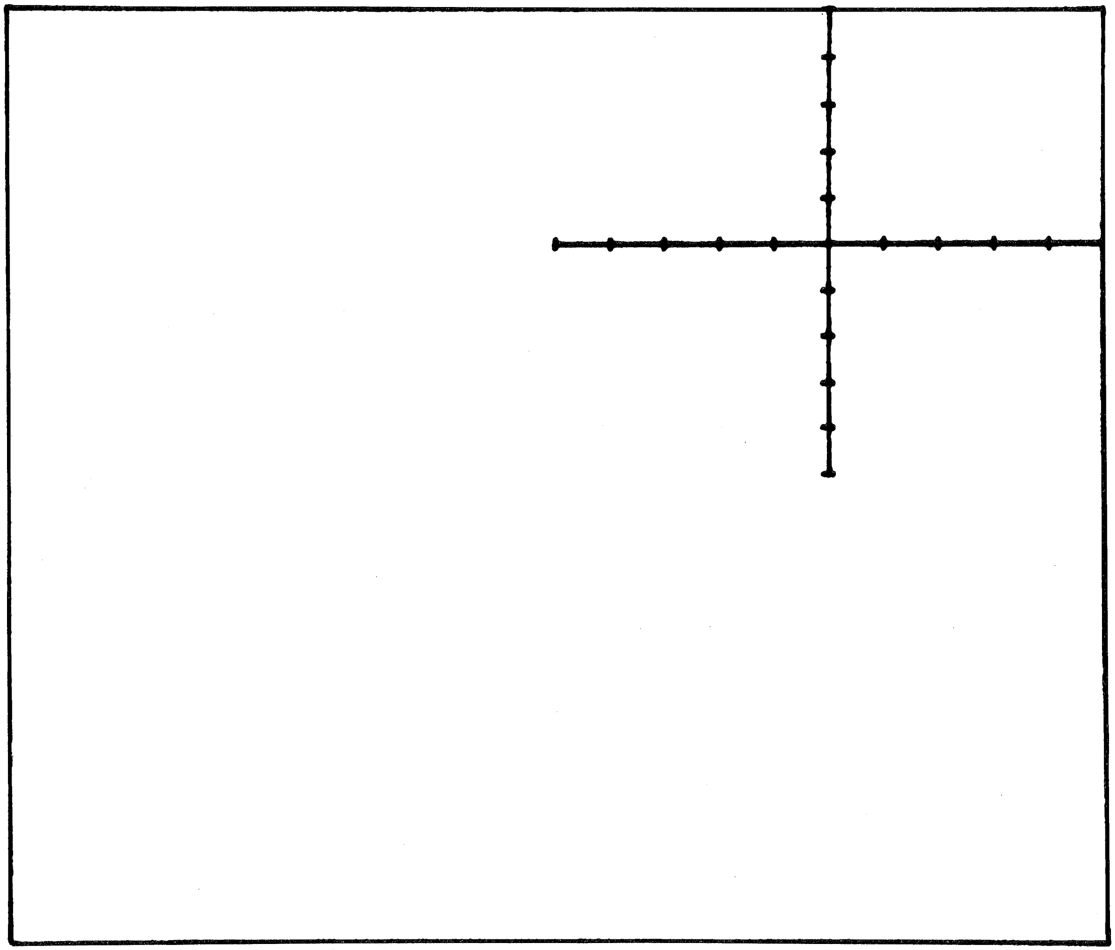


The axes cross at midscreen because of the range of the WINDOW statement. Tic marks are not drawn on the X axis because zero is specified as the first parameter. Tic marks on the vertical (Y) axis occur at intervals of 10 due to the second parameter specification of 10.

Another example:

```
100 PAGE
110 INIT
120 VIEWPORT 65,130,50,100
130 WINDOW -50,50,-50,50
140 AXIS 10,10
150 END
```

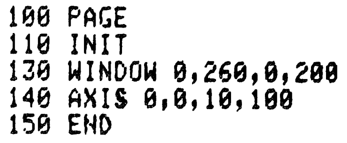
If you run the above program, the output is produced as shown below:



Only the upper right corner of the screen contains the plot because of the VIEWPORT parameters specified in line 120. The tic marks occur at 10 unit intervals because of the AXIS parameters specified in line 140. A further thing to be noted here is the length of the tic marks; they are smaller than the ones that result when you utilize the VIEWPORT and WINDOW parameters. The reason for this is that the length of the tic marks is always 1% of the viewport size in the corresponding direction. This means that the tic marks are the same length on both axes only when you use a "square" viewport like VIEWPORT 0,100,0,100 or something similar.

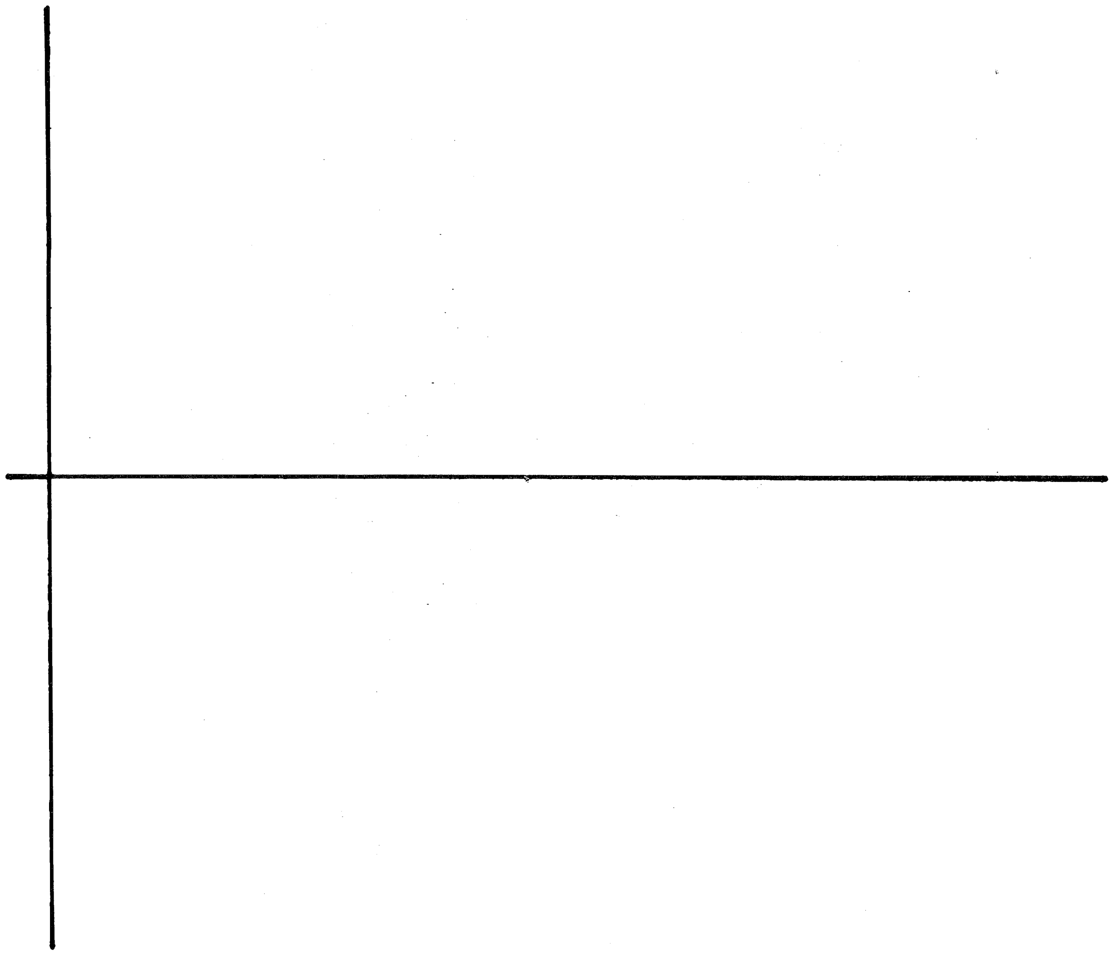
The AXIS Statement With Four Parameters

The third and fourth parameters of the AXIS statement allow you to manipulate the points where the axes intercept. The third parameter specifies the point where the vertical axis intercepts the horizontal axis. The fourth parameter specifies the point where the horizontal axis intercepts the vertical axis. These two points are specified in terms of user data units. For instance:



```
100 PAGE
110 INIT
130 WINDOW 0,260,0,200
140 AXIS 0,0,10,100
150 END
```

The output produced by the above example will look like the following:



The vertical axis crosses the horizontal axis of the point where X = 10. Similarly, the horizontal axis crosses the vertical axis where Y = 100.

At this point the AXIS statement has been discussed sufficiently for the syntax form to be presented:

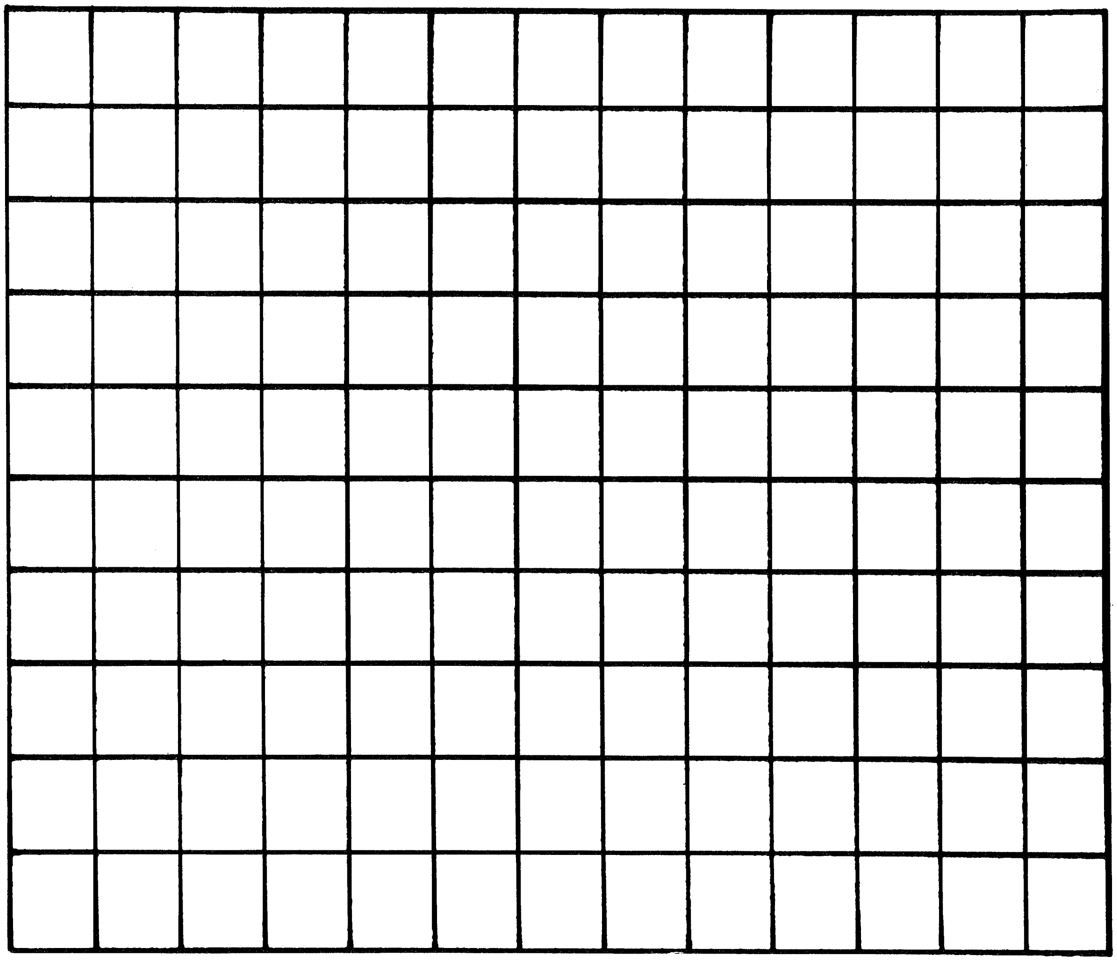
```
[Line number] AXIS [numeric expression, numeric expression
  [, numeric expression, numeric expression] ]
```

The first and second parameters are numeric expressions which determine the tic intervals in terms of user data units. The third and fourth parameters are also numeric expressions; they determine the axis intercept points. The default values for the first two parameters are zero, resulting in the suppression of the tic marks. The default values of the third and fourth parameters are zero if zero is within the window for the corresponding axis; otherwise the default values are set equal to the algebraic minimum data value of the corresponding axis.

In addition to the fact that this statement can be used to generate axis lines, it can also be used to generate grid lines for your graph. For example:

```
100 PAGE
110 INIT
120 FOR I=0 TO 130 STEP 10
130 AXIS 0,0,I,I
140 NEXT I
150 END
```

This program generates output as indicated below:



Notice that the program generates axis intercepts which are outside the WINDOW on the vertical axis. (The vertical axis goes to 100, the loop generates intercepts up to 130.) This is permissible; all that happens is that the "extra" axis lines are clipped and not drawn. An error message is not generated when this happens.

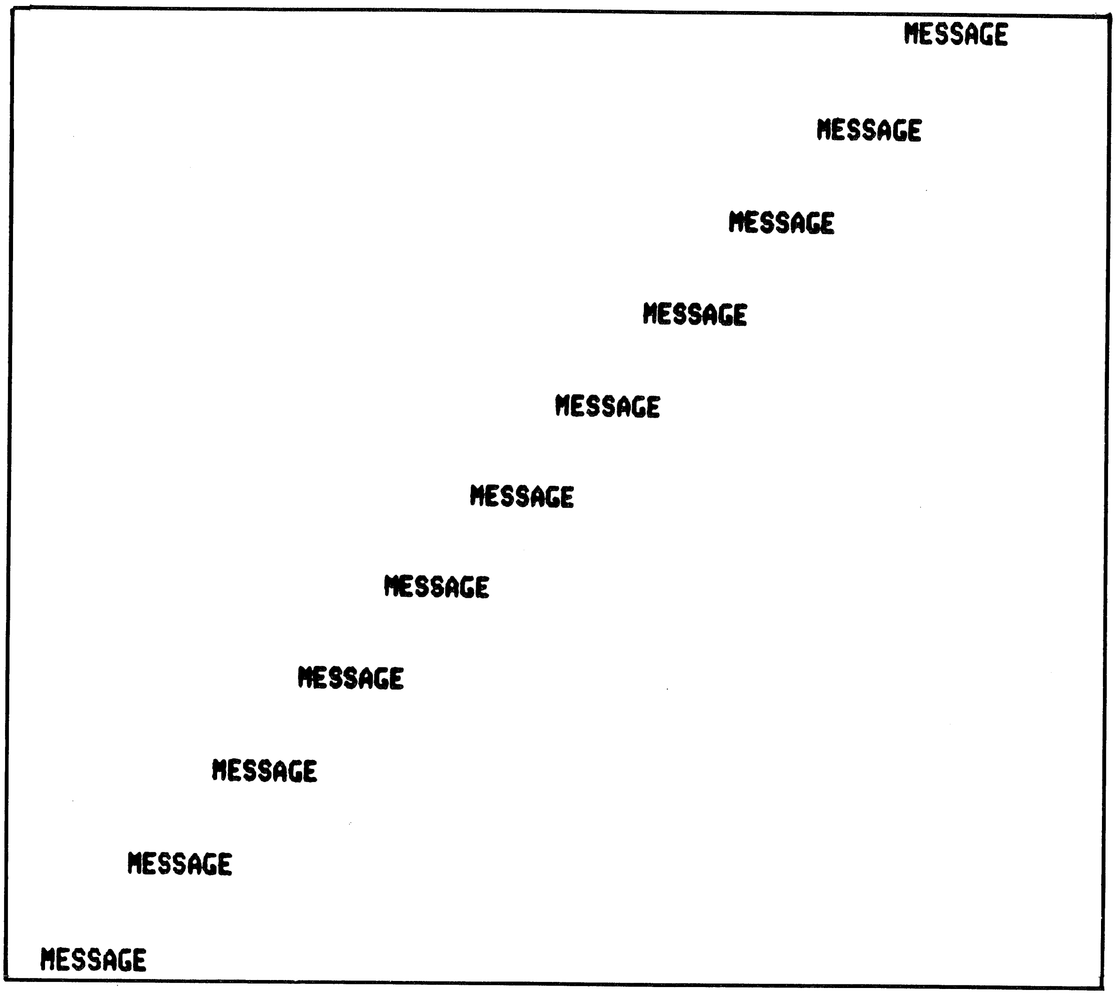
ALPHANUMERIC OUTPUT

The Graphic System gives you the ability to output alphanumeric information to any point on the screen. This becomes useful when you want to label the axes of a plot, print descriptive messages, add titles, etc. This is done by combining the familiar PRINT statement with various positioning techniques. The thing to keep in mind is that the location of the alphanumeric cursor determines the starting point for both alphanumeric and graphic output.

A MOVE statement can precede a PRINT statement to place alphanumeric information anywhere on the screen.

For example:

```
100 PAGE
110 INIT
120 LET A$="MESSAGE"
130 FOR I=0 TO 100 STEP 10
140 MOVE I,I
150 PRINT A$;
160 NEXT I
170 END
```



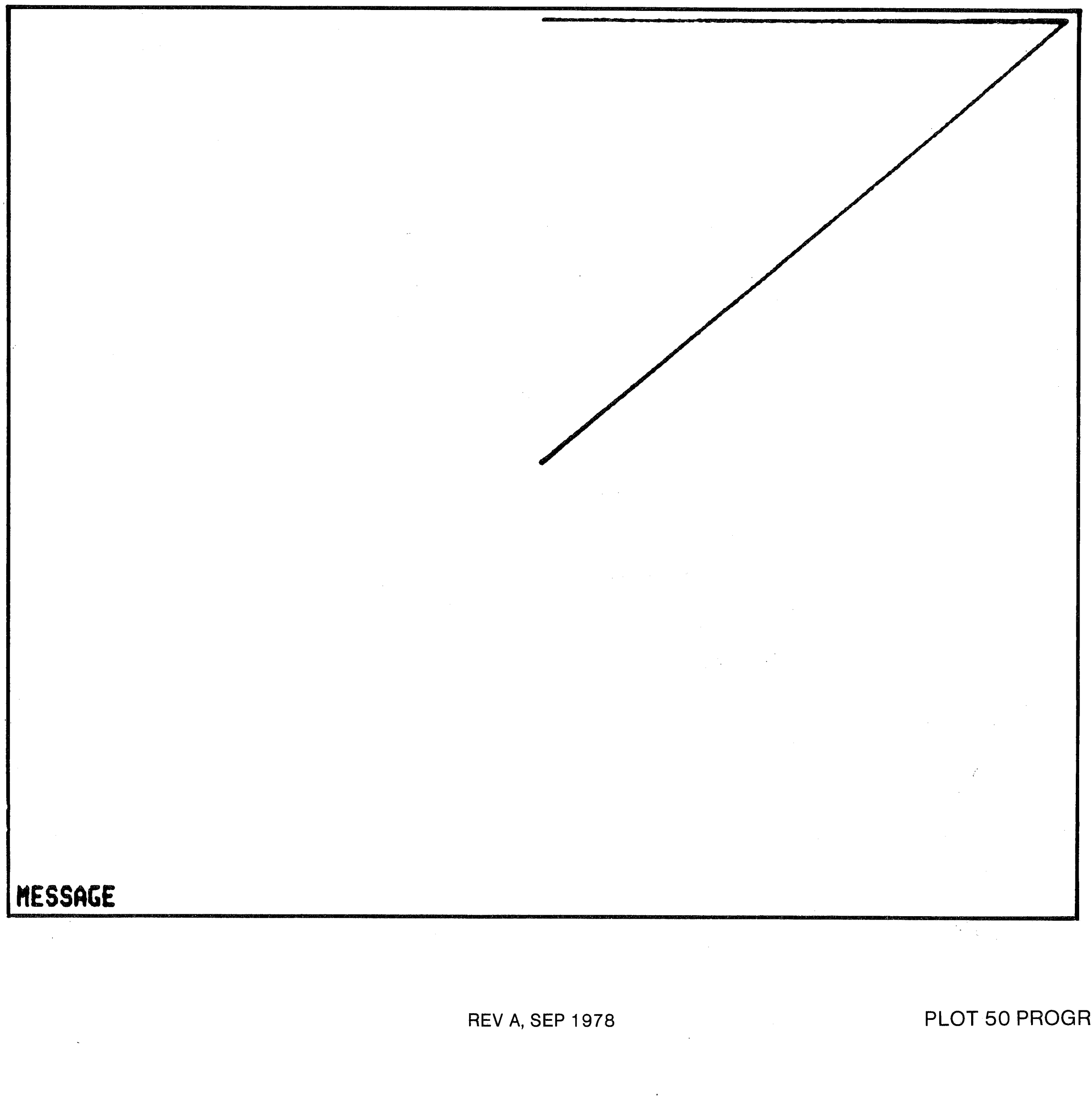
```
                                                                                          MESSAGE

                                                                                  MESSAGE

                                                                          MESSAGE

                                                                  MESSAGE

                                                          MESSAGE

                                                  MESSAGE

                                          MESSAGE

                                  MESSAGE

                          MESSAGE

                  MESSAGE

          MESSAGE
```

The above program generates a diagonal composed of the word "MESSAGE". After execution, the cursor remains at the next character location following the last "MESSAGE".

An additional example:

```
100 INIT
110 VIEWPORT 65,130,50,100
120 MOVE 0,100
130 DRAW 130,100
140 DRAW -65,-50
150 MOVE -130,-100
160 PRINT "MESSAGE";
170 END
```

The output looks like the following:

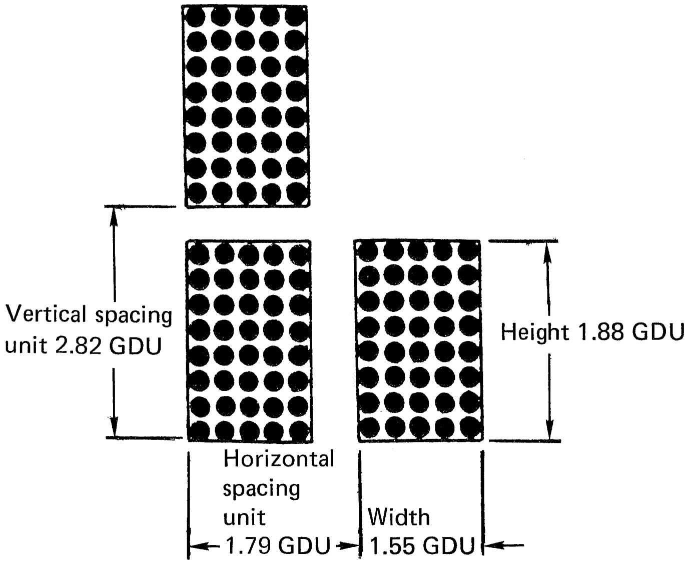


```
MESSAGE
```

Notice that the DRAW at line 130 is clipped; the MOVE at line 120 places the cursor at the upper left corner of the screen, and only that portion of the line within the viewport is drawn. Similarly, the DRAW in line 140 is clipped because the point to be drawn to is outside the viewport and outside the window. Again, only the portion of the line within the viewport is drawn. The MOVE at line 150, however, is not clipped, and neither is the PRINT.

The character size on a Graphic System display with an 11 inch crt is constant in size and orientation, regardless of the WINDOW, SCALE, and VIEWPORT parameters. The dimensions involved are shown below.

CHARACTER SIZE AND ORIENTATION


Vertical spacing unit 2.82 GDU
Height 1.88 GDU
Horizontal spacing unit 1.79 GDU
Width 1.55 GDU

In addition to the MOVE statement, you can utilize a set of special characters (called control characters) and the SPACE BAR to move the cursor around. These characters allow you to move the cursor horizontally and vertically in increments which are equal to the vertical and horizontal character spacing units specified in the above diagram. These characters, and their effects, are tabulated on the following page.

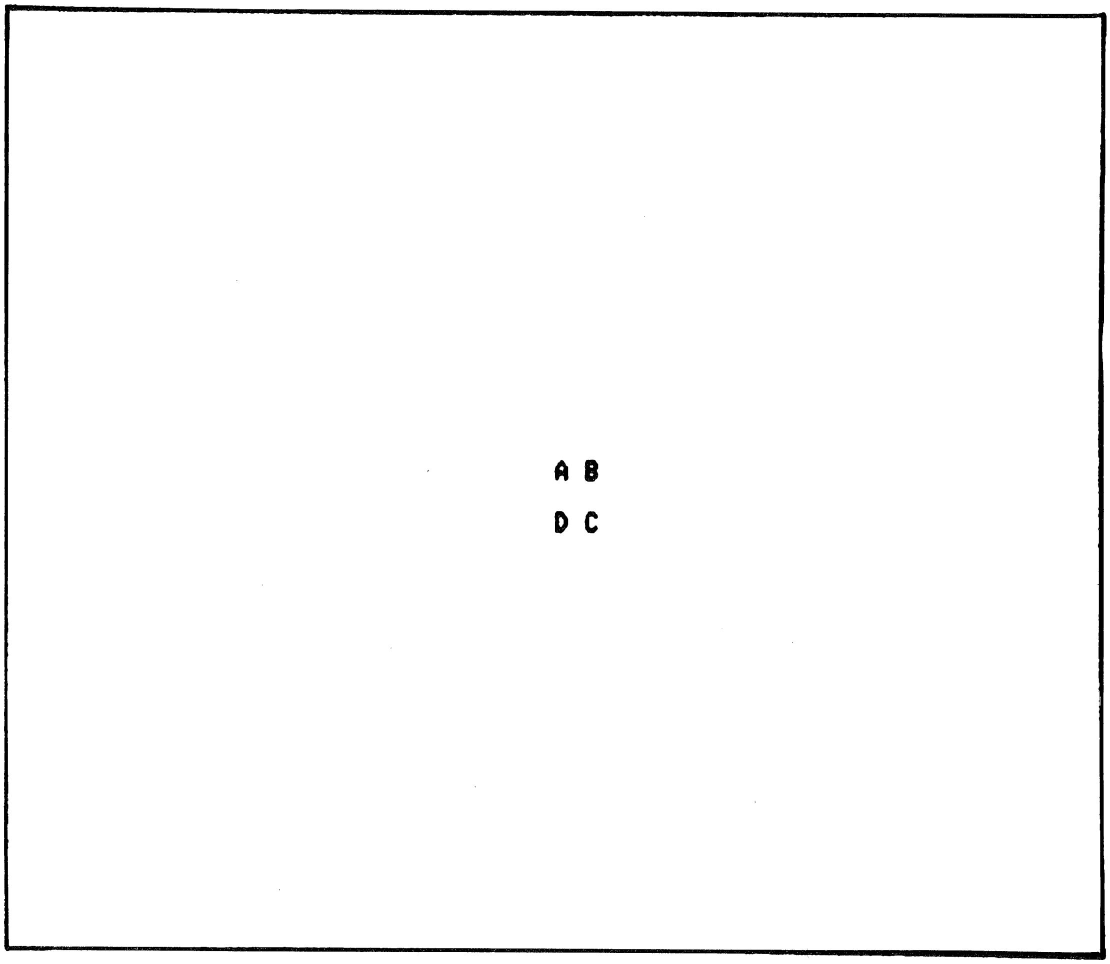
Character	Keys Used	Printout	Effect When Printed
Linefeed	CTRL J	J	Moves cursor down one vertical spacing unit.
Vertical tab	CTRL K	K	Moves cursor up one vertical spacing unit.
Backspace	CTRL H	H	Moves cursor left one horizontal spacing unit.
Space	SPACE BAR	SPACE	Moves cursor right one horizontal spacing unit.

The three control characters are entered in to the line buffer by simultaneously pressing down the CTRL (Control) key and the appropriate character key. The "PRINTOUT" column indicates the appearance of the characters on the screen.

Try running the following program to gain a feel for the use of these characters:

```
100 INIT
110 MOVE 65,50
120 PRINT "A BJJHCHHHDHKK";
130 END
```

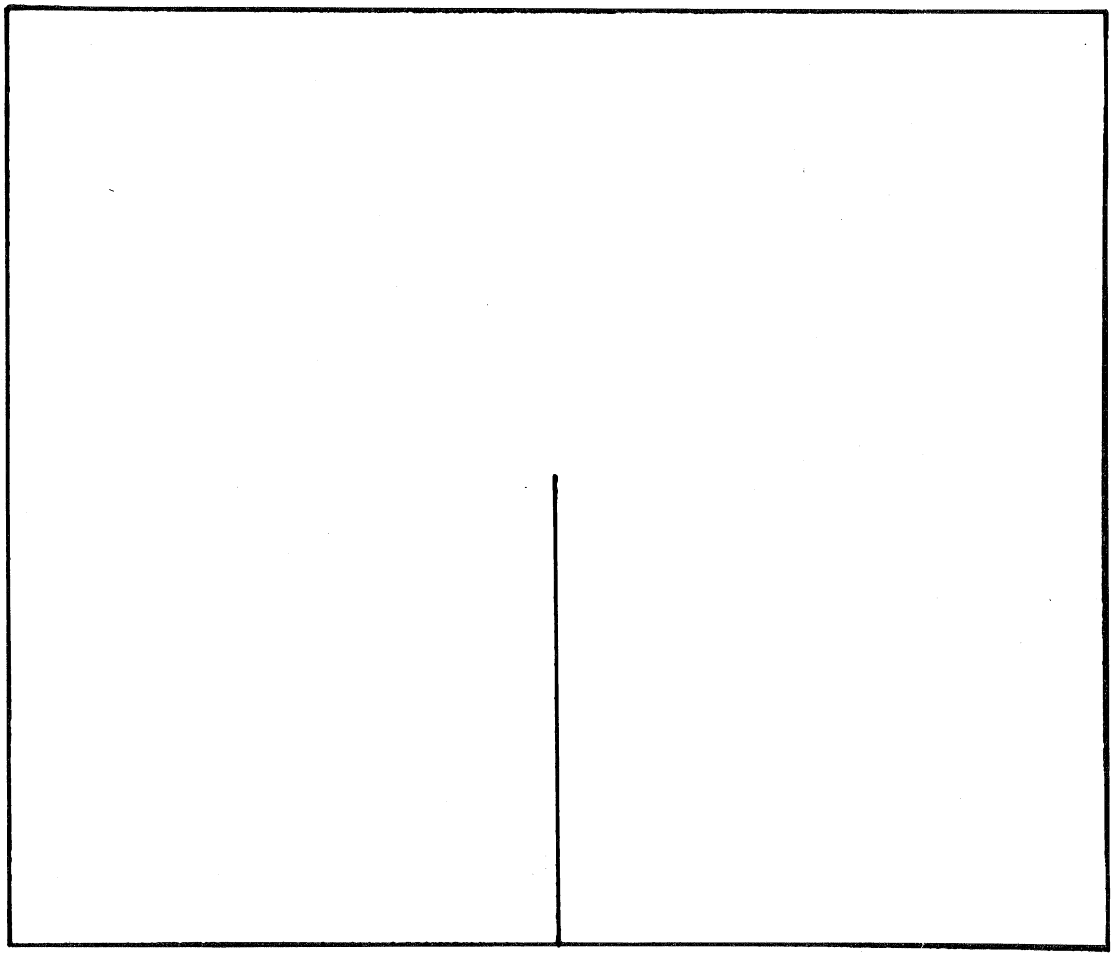
The result of the above program is that a square of characters appears, centered on the screen. The Graphic System does not print the control characters; only the "non-control" characters are printed. The control characters do, however, effect the location of the cursor. The program terminates with the cursor located in the same position as the "A".



It was mentioned earlier that the cursor determines the starting point for both alphanumeric output and graphic output. This fact is demonstrated by the following program:

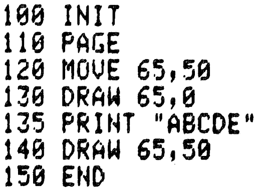
```
100 INIT
110 PAGE
120 MOVE 65,50
130 DRAW 65,0
140 DRAW 65,50
150 END
```

The effect of the above program is that a vertical line appears centered in the lower half of the screen. The line is actually drawn twice; but of course, you see only one line.



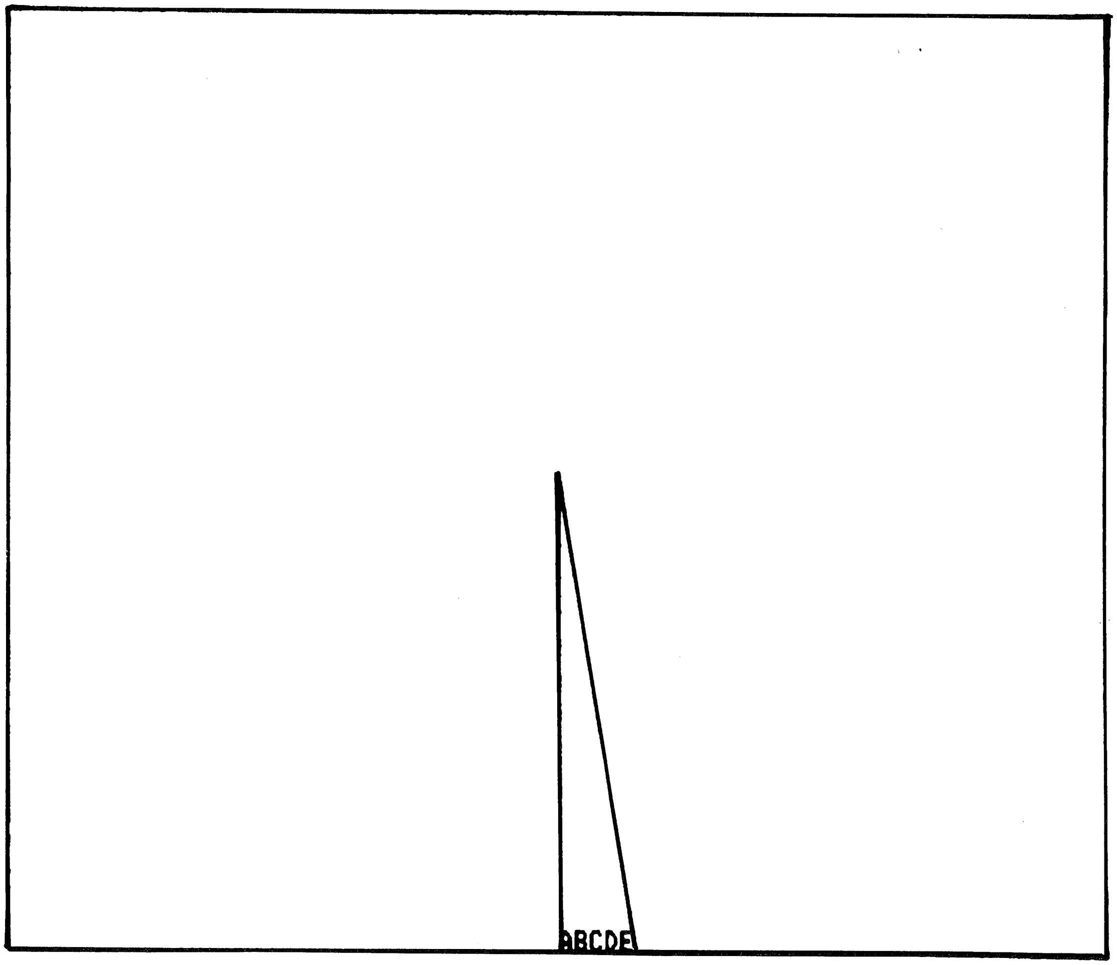
Now, to illustrate the effect of the cursor in determining the starting point for graphic output, key in the following addition to the program:

135 PRINT "ABCDE"



```
100 INIT
110 PAGE
120 MOVE 65,50
130 DRAW 65,0
135 PRINT "ABCDE";
140 DRAW 65,50
150 END
```

Run the program again, and observe the result shown below:

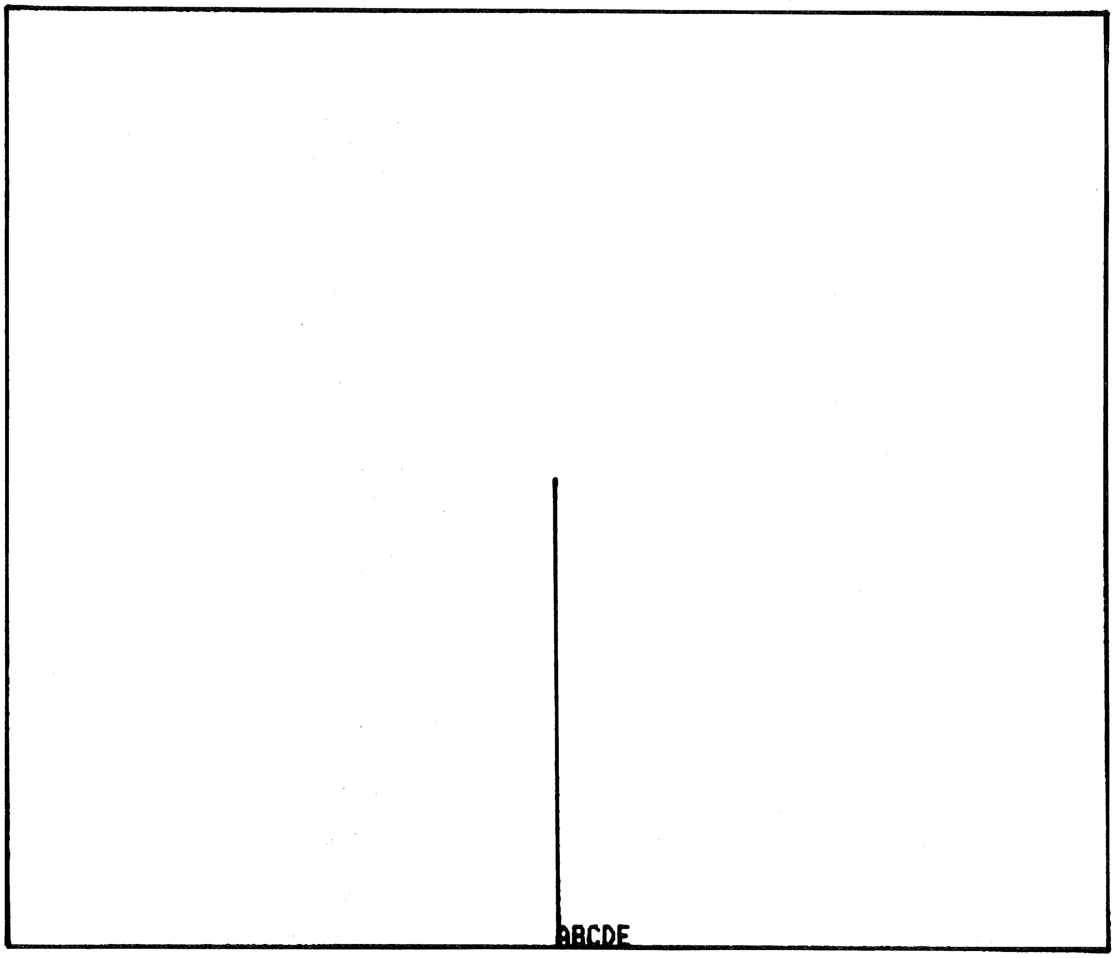


An offset is produced by the PRINT statement. The cursor is displaced, and now you can see two lines instead of one.

This effect can be avoided by keeping track of the number of characters that are output, and then relocating the cursor through the use of the control characters described earlier. For example:

```
100 INIT
110 PAGE
120 A$="ABCDE"
130 MOVE 65,50
140 DRAW 65,0
150 PRINT A$;
160 FOR I=1 TO LEN(A$)
170 PRINT "H̲";
180 NEXT I
190 DRAW 65,50
200 END
```

If you run the above program, the output appears as follows:



The two lines are drawn in the same location again. Lines 160 through 180 output sufficient backspace characters to position the cursor (and the graphic point) back at the bottom of the line.

SUMMARY

The fundamental building block for graphics is the straight line. Straight lines are defined by two end points. These end points are specified by horizontal and vertical coordinates.

INIT initializes the system so that the screen is regarded as 130 units wide by 100 units high. PAGE clears the screen and sends the cursor to "home" (the upper left corner of the screen).

You can position the cursor without producing a visible line with the MOVE statement. Visible lines result from the DRAW statement. INIT followed by MOVE 65,50 positions the cursor at mid-screen.

WINDOW refers to the "user data space" and specifies the bounds of a two-dimensional plane. User data units refer to pounds, centimeters, furlongs, or any unit of measure you wish to work with. GDU stands for Graphic Display Units, and refers to the units of measurement that the system works with internally. VIEWPORT defines the drawing boundaries for graphic output on the screen. The transformation process for converting user data units to GDU's is called mapping and is done automatically by the system.

WINDOW defines the limits of the user data space which is mapped into the viewport. These limits, in conjunction with the selected VIEWPORT, imply a scale factor. SCALE, on the other hand, explicitly defines the scale factor and this implies a certain WINDOW size. SCALE sets the location 0,0 to be the location of the graphic point just before SCALE is executed.

MOVE and DRAW specify the "absolute" location of the finish point along each axis. RMOVE and RDRAW specify the horizontal and vertical intervals away from the present location of the cursor for the finish point, the intervals are "relative".

ROTATE applies to relative moves and draws. The point of rotation is the finish point of the last absolute move or draw. ROTATE — n yields clockwise rotation; ROTATE n yields counterclockwise rotation.

Data stored in arrays is plotted using MOVE, DRAW, RMOVE, and RDRAW. DRAW X,Y causes the system to draw to the coordinates X(1), Y(1), X(2), Y(2), and so on. Two dimensional arrays are used similarly with the output in row major order.

Axis lines are produced by the AXIS statement. This statement allows you to specify horizontal and vertical intercepts and tic intervals. You can also generate grid lines with AXIS.

Alphanumeric information can be placed anywhere on the screen with combinations of MOVE (or RMOVE) and PRINT. However, alphanumeric output obviously affects the location of the graphic point, and this may require special attention.

EXAMPLE PROGRAMS

TITLE: Histogram

DESCRIPTION: This is a simple histogram program which inputs data, and then outputs a histogram representing the data. The program also outputs horizontal and vertical axes, with tic marks on the vertical axis.

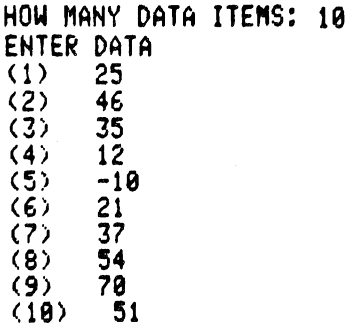
PROGRAM LISTING:

```
100 REMARK HISTOGRAM
110 PAGE
120 INIT
130 PRINT "HOW MANY DATA ITEMS: ";
140 INPUT A
150 DELETE B
160 DIM B(A)
170 REMARK INITIALIZE VARIABLES TO EXTREME VALUES FOR MIN-MAX SCAN
180 M1=1.0E+300
190 M2=-1.0E+300
200 PRINT "ENTER DATA"
210 REMARK LOOP TO INPUT DATA AND OBTAIN MIN-MAX DATA VALUES
220 FOR I=1 TO A
230 PRINT "(";I;")    ";
240 INPUT B(I)
250 REMARK M1 IS MIN, M2 IS MAX
260 M1=B(I) MIN M1
270 M2=B(I) MAX M2
280 NEXT I
290 PAGE
300 REMARK DRAW HISTOGRAM
310 VIEWPORT 15,130,10,100
320 WINDOW 0,A,M1,M2
330 AXIS 0,(M2-M1)/10
340 FOR I=1 TO A
350 MOVE I-1,0
360 RDRAW 0,B(I)
370 RDRAW 1,0
380 RDRAW 0,-B(I)
390 NEXT I
400 REMARK INCREASE VIEWPORT AND LABEL MIN-MAX POINTS
410 VIEWPORT 0,130,10,100
420 MOVE 0,M1
430 PRINT M1;
440 MOVE 0,M2
450 PRINT M2;
460 REMARK INCREASE VIEWPORT FURTHER TO PRINT TIC INTERVAL INFORMATION
470 VIEWPORT 0,130,0,100
480 MOVE 0,M1
490 PRINT "VERTICAL AXIS TIC INTERVAL= ";(M2-M1)/10;
500 HOME
510 END
```

METHODOLOGY: The array that contains the data is dimensioned in line 160 to correspond to the number of data items. The FOR/NEXT loop of lines 220 through 280 inputs the data and, at the same time, obtains the minimum and maximum data values. (The minimum and maximum values are used later in the program to establish the WINDOW parameters.)

The VIEWPORT statement in line 310 establishes a margin on the left and bottom edges of the screen to provide room for labeling. Lines 340 through 390 include a FOR/NEXT loop which outputs the histogram. Lines 430 and 450 label the minimum and maximum data values on the vertical axis, and line 490 outputs the tic interval information.

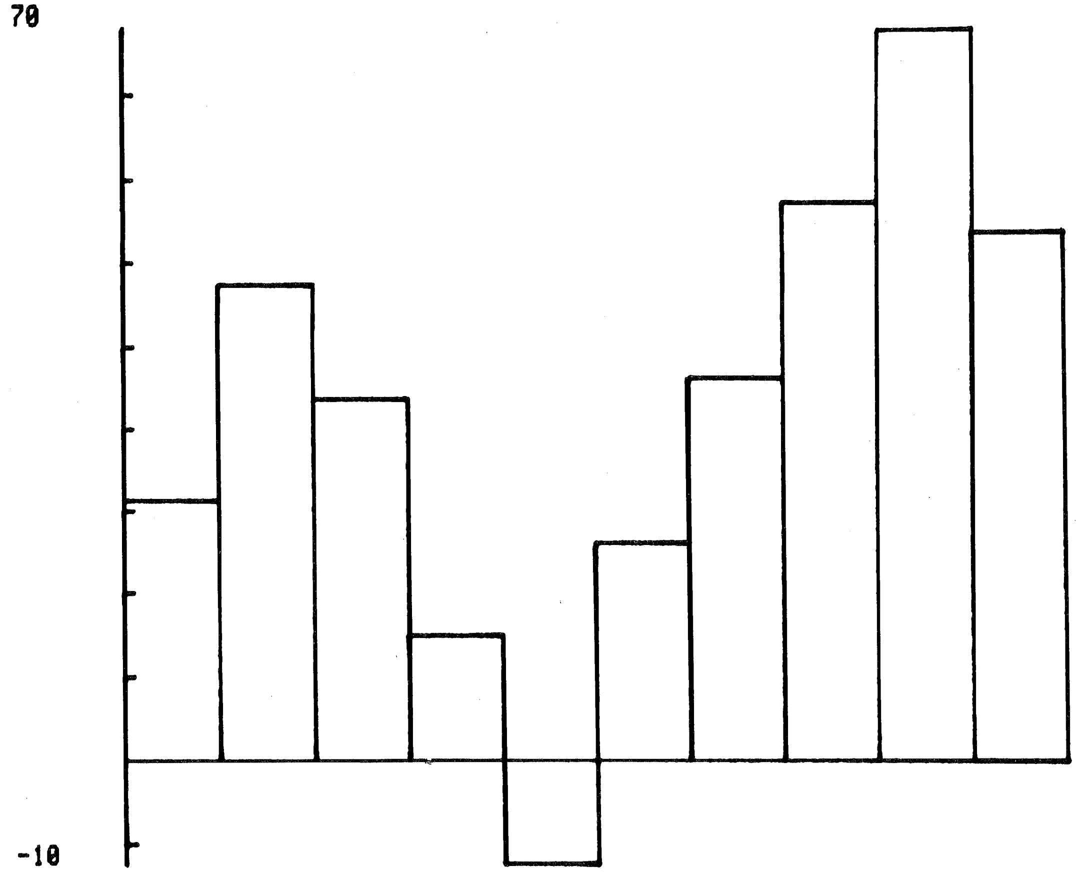
OPERATING PROCEDURE: Type RUN, press RETURN, and the program responds with "HOW MANY DATA ITEMS?" Enter the appropriate number and press RETURN. The program then says "ENTER DATA" and provides guide numbers enclosed in parentheses to indicate the number of the data item being input. You follow each data item with a RETURN character, causing the next guide number to appear. This continues until all the data items have been input. The following example illustrates this process:



```
HOW MANY DATA ITEMS: 10
ENTER DATA
(1)   25
(2)   46
(3)   35
(4)   12
(5)   -10
(6)   21
(7)   37
(8)   54
(9)   70
(10)   51
```

After the last data item has been entered, a PAGE statement clears the screen, and the histogram is then generated.

OUTPUT SAMPLE:



```
VERTICAL AXIS TIC INTERVAL= 8
```

TITLE: **Y Only Data Plot**

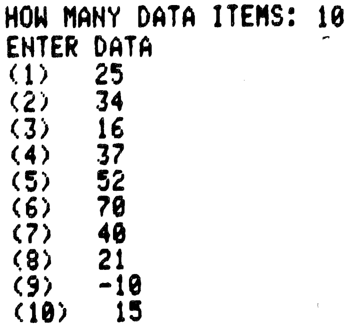
DESCRIPTION: This program inputs data and outputs a simple Y only data plot of the data. The program is essentially the same as the previous histogram program.

PROGRAM LISTING:

```
100 REMARK Y ONLY DATA PLOT
110 PAGE
120 INIT
130 PRINT "HOW MANY DATA ITEMS: ";
140 INPUT A
150 DELETE B
160 DIM B(A)
170 REMARK INITIALIZE VARIABLES TO EXTREME VALUES FOR MIN-MAX SCAN
180 M1=1.0E+300
190 M2=-1.0E+300
200 PRINT "ENTER DATA"
210 REMARK LOOP TO INPUT DATA AND OBTAIN MIN-MAX DATA VALUES
220 FOR I=1 TO A
230 PRINT "(";I;")    ";
240 INPUT B(I)
250 REMARK M1 IS MIN, M2 IS MAX
260 M1=B(I) MIN M1
270 M2=B(I) MAX M2
280 NEXT I
290 PAGE
300 REMARK DRAW Y ONLY DATA PLOT
310 VIEWPORT 15,130,10,90
320 WINDOW 0,A,M1,M2
330 AXIS 1,(M2-M1)/10
340 MOVE 1,B(1)
350 FOR I=1 TO A
360 DRAW I,B(I)
370 NEXT I
380 REMARK INCREASE VIEWPORT AND LABEL MIN-MAX POINTS
390 VIEWPORT 0,130,10,90
400 MOVE 0,M1
410 PRINT M1;
420 MOVE 0,M2
430 PRINT M2;
440 REMARK INCREASE VIEWPORT FURTHER TO PRINT TIC INTERVAL INFORMATION
450 VIEWPORT 0,130,0,100
460 MOVE 0,M1
470 PRINT "VERTICAL AXIS TIC INTERVAL= ";(M2-M1)/10;
480 MOVE 0,M2
490 PRINT "Y-ONLY DATA PLOT"
500 HOME
510 END
```

METHODOLOGY: The only significant difference between this program and the previous histogram program is found in FOR/NEXT loop of lines 350 through 370. Instead of drawing "cells" to represent the data, the program draws to the location corresponding to the "next" data point. The horizontal ("X") axis coordinates are obtained from the FOR/NEXT variable, and the vertical ("Y") axis coordinates are obtained from the data.

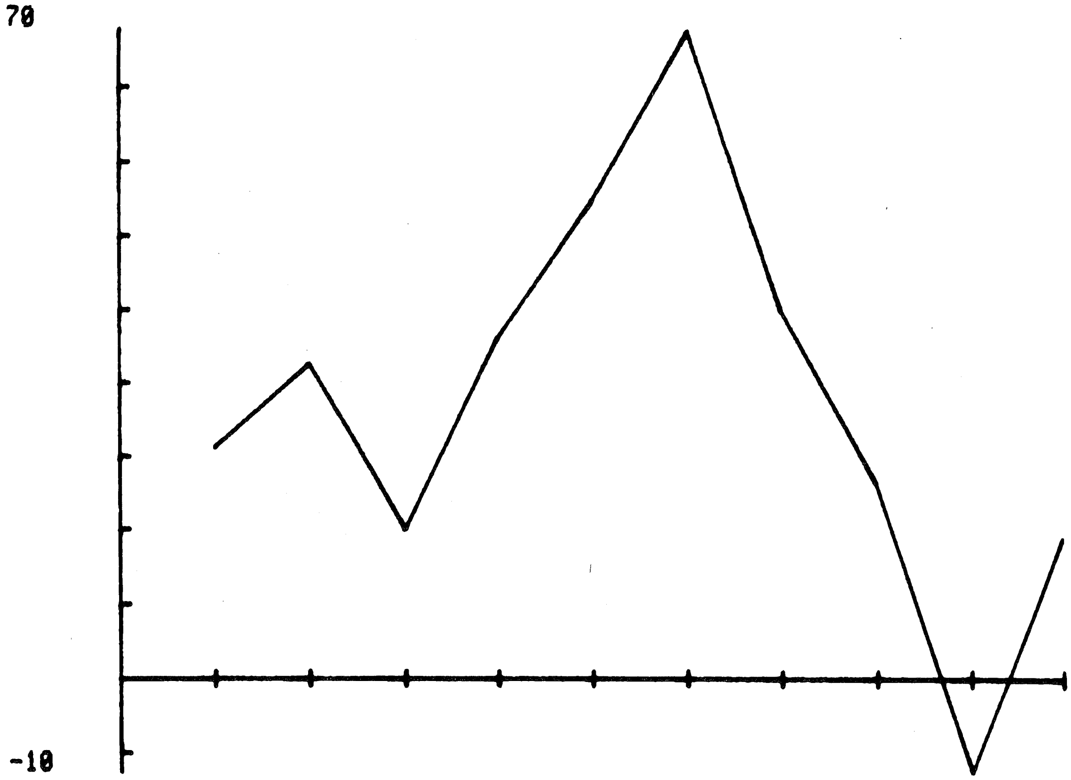
OPERATING PROCEDURE: Type RUN, press RETURN, and the program responds with "HOW MANY DATA ITEMS:". Enter the appropriate number and press RETURN. The next prompting is "ENTER DATA"; you then enter the data in the same way that the histogram program inputs data. This is illustrated below:



```
HOW MANY DATA ITEMS: 10
ENTER DATA
(1)   25
(2)   34
(3)   16
(4)   37
(5)   52
(6)   70
(7)   40
(8)   21
(9)   -10
(10)   15
```

OUTPUT SAMPLE:

Y-ONLY DATA PLOT



VERTICAL AXIS TIC INTERVAL= 8

TITLE: Random Polygons

DESCRIPTION: This is primarily an "amusement" program that generates interesting patterns by drawing "random" polygons. You can specify the number of sides the polygon has, and the number of frames that are generated.

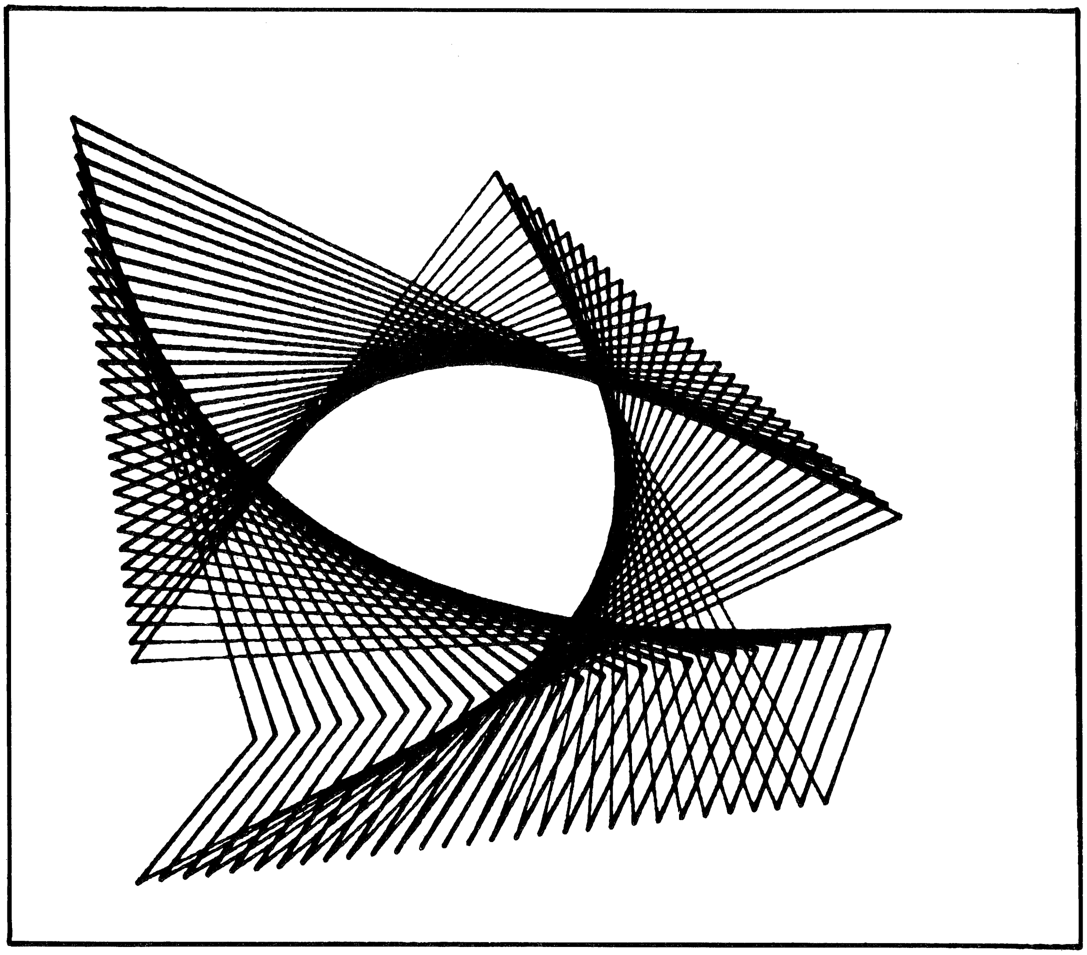
PROGRAM LISTING:

```
4 GO TO 100
100 INIT
110 PAGE
120 PRINT "ENTER NUMBER OF SIDES ";
130 INPUT A
140 PRINT "ENTER NUMBER OF FRAMES ";
150 INPUT B
160 DIM C(A,2),D(A,2)
170 REMARK GENERATE INITIAL "COORDINATE" ARRAY
180 FOR I=1 TO A
190 C(I,1)=RND(-1)*130
200 C(I,2)=RND(-1)*100
210 NEXT I
220 REMARK GENERATE "INCREMENT" ARRAY
230 FOR I=1 TO A
240 D(I,1)=(RND(-1)*130-C(I,1))/B
250 D(I,2)=(RND(-1)*100-C(I,2))/B
260 NEXT I
270 PAGE
280 FOR J=1 TO B
290 MOVE C(A,1),C(A,2)
300 REMARK DRAW POLYGON
310 FOR I=1 TO A
320 DRAW C(I,1),C(I,2)
330 NEXT I
340 REMARK ADD "INCREMENT" ARRAY TO "COORDINATE" ARRAY
350 C=C+D
360 NEXT J
370 END
```

METHODOLOGY: The numeric variable A receives the number of sides of the polygon, and B receives the number of frames. Next, a "coordinate" array C and an "increment" array D are both dimensioned in line 160 to contain A rows and 2 columns. The 2 columns correspond to the "X" and "Y" coordinates for each side of the polygon. Next, a FOR/NEXT loop (lines 180 through 210) fills array C with random numbers for the coordinates of the first polygon that is drawn. Following this, another FOR/NEXT loop (lines 230 through 260) fills array D with random numbers that are later added to array C to become the coordinates of the "next" polygon. The nested loops of lines 280 through 360 draw the polygons. Notice in line 350 that array D (the "increment" array) is added to array C (the "coordinate" array) to become the coordinates for the next polygon.

OPERATING PROCEDURE: Press user definable key number 1. The program responds with "ENTER NUMBER OF SIDES". Enter the desired number of sides and press RETURN. (A small number in the range of 3 to 8 works most satisfactorily.) Next, the program asks "ENTER NUMBER OF FRAMES". Enter the number of frames you want and press RETURN. (A number in the range of 25—100 works well.) Following this, the random polygon pattern is generated.

OUTPUT SAMPLE: The output below resulted from specifying 4 sides and 30 frames.



Appendix A

REFERENCE MATERIAL

HIERARCHY OF OPERATIONS

The priority of the various Graphic System operations is summarized below. The highest priority operation is listed first, and the lowest priority operation is listed last. Equal priority operators within a BASIC statement are evaluated left-to-right.

Priority	Operation	Symbol or Keyword
1	Left parenthesis	(
2	Functions, including user-defined functions	SIN, COS, PI, etc.
3	Monadic + and –	+ and –
4	Exponentiation	↑
5	Multiplication and division	* and /
6	Dyadic + and –	+ and –
7	Minimum and maximum operators	MIN and MAX
8	Relational operators	=, <>, <, >, <=, >=
9	Negating logical operator	NOT
10	Logical opeators	AND, OR
11	Right parenthesis	)

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 A0 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.
Binary String	A connected sequence of 1's and 0's.
Bit	A Binary digit. A unit of data in the binary numbering system; a 1 or 0.
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".
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.

Term	Definition
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 dot matrix 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 program statements are entered into memory.
Delimiter	A character that fixes the limits, or bounds, or 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.
Flowchart	A programming tool that provides a graphic representation of a routine to solve a specific problem.
Function	A special purpose operation referring to a set of calculations within an expression, as in the sine function, square root function, etc.
Graphic Display Unit (GDU)	An internal unit of measure representing one one-hundredth of the vertical axis on the graphic drawing surface.
Graphics	Computer output that is composed of lines rather than letters numbers, and symbols.
Hardware	The physical devices and components of a computer.

Term	Definition
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(50); the indexes are 1, 2, . . . 50.
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.
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 0. 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 0's, specifically, the AND, OR, and NOT operators. "True" operations return "1", "false" operations return "0".
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.

Term	Definition
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+200 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 A0 through Z9, and can be used in numeric expressions.
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 –50, 50–100, 100, the parameters are –50, 50, –100, and 100.

Term	Definition
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.
Scientific Notation	A format representing numbers as a fractional part, or mantissa, and a power of 10, or characteristic, as in 1.23E45.
Software	Prepared programs that simplify computer operations, such as mathemetics and statistics software. Software must be reloaded 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.

Term	Definition
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.
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.
User Data Units	The units of measure the programmer elects to work with for a particular graphing application. These units are estabilished in the WINDOW statement as a numeric range for each axis. For example the vertical axis range can be set starting at 0 "dollars" and ending at 100 "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 A0 through Z9. String variables may be named with the characters A$ through Z$.

CONTROL CHARACTERS

The control characters listed below may be used within PRINT statements to obtain some useful effects, and also to save space in a program.

Keys Used	Printout	Effect
CTRL G	G	Sounds the bell.
CTRL H	H	Backspace
CTRL I	I	Tab
CTRL J	J	Linefeed
CTRL K	K	Vertical tab
CTRL L	L	Page and Home
CTRL ↑	↑	Home
CTRL RUBOUT	—	Carriage return, Line feed
SPACE		Moves cursor forward one space

EXAMPLES: To sound the bell:

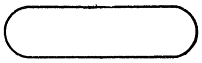
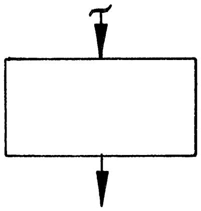
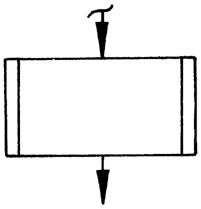
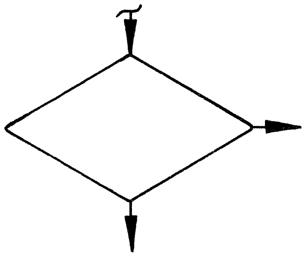
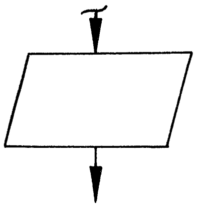
```
100 PRINT"G"
```

To print a line of text and follow it with two blank lines:

```
100 PRINT "LINE OF TEXTJJ"
110 PRINT "ANOTHER LINE OF TEXT"
RUN
LINE OF TEXT


ANOTHER LINE OF TEXT
```

FLOWCHART SYMBOLS

The following set of symbols is an abbreviated version of the various flowchart symbols in use today, and should be sufficient for most introductory applications.

Symbol	Meaning
	Terminal. This is used to indicate starting and ending points of a routine.
	Process. This might be something like "increment the counter", "initialize", etc.
	Predetermined process (subroutine). Rather than flowcharting the subroutine each time it is used, just flowchart it once and use the subroutine symbol to indicate its usage.
	Decision. Test to see if a condition is true. If it is, take one branch, otherwise take the other branch.
	Input/Output. This is a generalized symbol indicating an input or output operation.

ASCII CODE CHART

NUL (0)	DLE (16)	SP (32)	0 (48)	@ (64)	P (80)	\ (96)	p (112)
SOH (1)	DC1 (17)	! (33)	1 (49)	A (65)	Q (81)	a (97)	q (113)
STX (2)	DC2 (18)	" (34)	2 (50)	B (66)	R (82)	b (98)	r (114)
ETX (3)	DC3 (19)	# (35)	3 (51)	C (67)	S (83)	c (99)	s (115)
EOT (4)	DC4 (20)	$ (36)	4 (52)	D (68)	T (84)	d (100)	t (116)
ENQ (5)	NAK (21)	% (37)	5 (53)	E (69)	U (85)	e (101)	u (117)
ACK (6)	SYN (22)	& (38)	6 (54)	F (70)	V (86)	f (102)	v (118)
BEL (7)	ETB (23)	' (39)	7 (55)	G (71)	W (87)	g (103)	w (119)
BS (8)	CAN (24)	((40)	8 (56)	H (72)	X (88)	h (104)	x (120)
HT (9)	EM (25)	) (41)	9 (57)	I (73)	Y (89)	i (105)	y (121)
LF (10)	SUB (26)	* (42)	: (58)	J (74)	Z (90)	j (106)	z (122)
VT (11)	ESC (27)	+ (43)	; (59)	K (75)	[(91)	k (107)	{ (123)
FF (12)	FS (28)	, (44)	< (60)	L (76)	\ (92)	l (108)	¦ (124)
CR (13)	GS (29)	– (45)	= (61)	M (77)	] (93)	m (109)	} (125)
SO (14)	RS (30)	. (46)	> (62)	N (78)	∧ (94)	n (110)	~ (126)
SI (15)	US (31)	/ (47)	? (63)	O (79)	_ (95)	o (111)	RUBOUT (DEL) (127)

Note

During default (SET CASE) string comparisons, lower case characters (gray tint) are converted to their upper case equivalents.

ERROR MESSAGES

Message Number	Error Message
1	An arithmetic operation has resulted in an out of range number. Example: 1/1.0E-308
2	A divide by zero operation has resulted in an out of range number. Example: 4/0
3	An exponentiation operation has resulted in an out of range number. Example: 5↑1.0E+300
4	An exponentiation operation involving the base e has resulted in an out of range number. Example: EXP (1.0E+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
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

Message Number	Error Message
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 300
14	A parameter for a CALL 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 0
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	Not used.

Message Number	Error Message
19	There is an invalid parameter in a GOTO, FOR, or NEXT statement. Example: 500 FOR I=1 to 20 where I has been previously defined as an array variable.
20	The logical unit number specified in the statement is not within the range 0 through 9. Example: 100 ON EOF (10) THEN 500
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 0 and the exponent is not an integer less than 256. Example: −10↑257.5
23	An attempt has been made to take the LOG or LGT of a number which is equal to or less than 0. Example: LOG (−1)
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 0 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.

Message Number	Error Message
29	The parameter in the VAL function is not a character string containing a valid number. Example: A=VAL("Hi")
30	The matrix multiply operation failed because the target variable has the same name as one of the parameters.
31	The matrix inversion failed because the determinent was 0. This error is treated as a SIZE error.
32	The routine name specified in the CALL statement is misspelled or can not be found. 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.
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.

Message Number	Error Message
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 SIZE interrupt condition has occurred and an ON SIZE THEN statement has not been executed in the current BASIC program.
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.
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 500 where the line 500 doesn't exist or PRINT USING 100: where line 100 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.

Message Number	Error Message
53	After 10 attempts, the internal magnetic tape unit has been unable to read a portion of the current magnetic tape. The tape head has been positioned after the bad portion in the file to allow the rest of the file to be read.
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 an OLD statement when the tape head is positioned in the middle of a data file.
56	An attempt has been made to send information to a write-protected tape. Remove the tape cartridge, rotate the lock-out plug 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.
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 Disk 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 0 through 255.

Message Number	Error Message
67	An attempt has been made to execute an illegal I/O operation on an internal peripheral device. Example: DRAW @33:50,50
68	The diagnostic loader failed.
69	An input error or an output error has occured 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: 100 IMAGE 6D,5("MARK
71	A format string is not specified for the PRINT USING operation. Example: 100 IMAGE 6D 110 PRINT USING 100: 23,24,25 Line 100 should be: 100 IMAGE 3(6D)
73	There is an invalid character in the format string specified in the PRINT USING statement.
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: 100 IMAGE 3(6D) 110 PRINT USING 100:A,B Line 100 should be: 100 IMAGE 2(6D)
76	Parantheses are incorrectly used in the format string which is specified in the PRINT USING statement. Example: 100 IMAGE 2(6D 110 PRINT USING 100:A,B Line 100 should be 100 IMAGE 2(6D)

Message Number	Error Message
77	There is an invalid modifier to a field operator in the format string which is specified in the PRINT USING statement. Example: 100 IMAGE 2(6D),2S 110 PRINT USING 100:A,B Line 100 should be: 100 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: 100 IMAGE 4D,S,8A Line 100 should be: 100 IMAGE 4D,8A,S
79	A comma is incorrectly used in the format string which is specified in the PRINT USING statement. Example: 100 IMAGE 6,D,S Line 100 should be: 100 IMAGE 6D,S
80	A decimal point is incorrectly used in the format string which is specified in the PRINT USING statement. Example: 100 IMAGE .3D 110 PRINT USING 100:812.345 Line 100 should be: 100 IMAGE FD.3D
81	A data type mismatch has occurred in the PRINT USING statement. Example: 100 IMAGE 6D,6A 110 PRINT USING 100: "MARY",26 Line 100 should be: 100 IMAGE 6A,6D
82	A tabbing error has occurred in the format string which is specified in the PRINT USING statement. Example: 100 IMAGE 10A,2T,FD 110 PRINT USING 100: "ENTER DATA",D The absolute tab to position 2 specified by 2T in line 100 cannot occur because the cursor has already advanced beyond position 2. The tab specification must be at least 11T in this case.

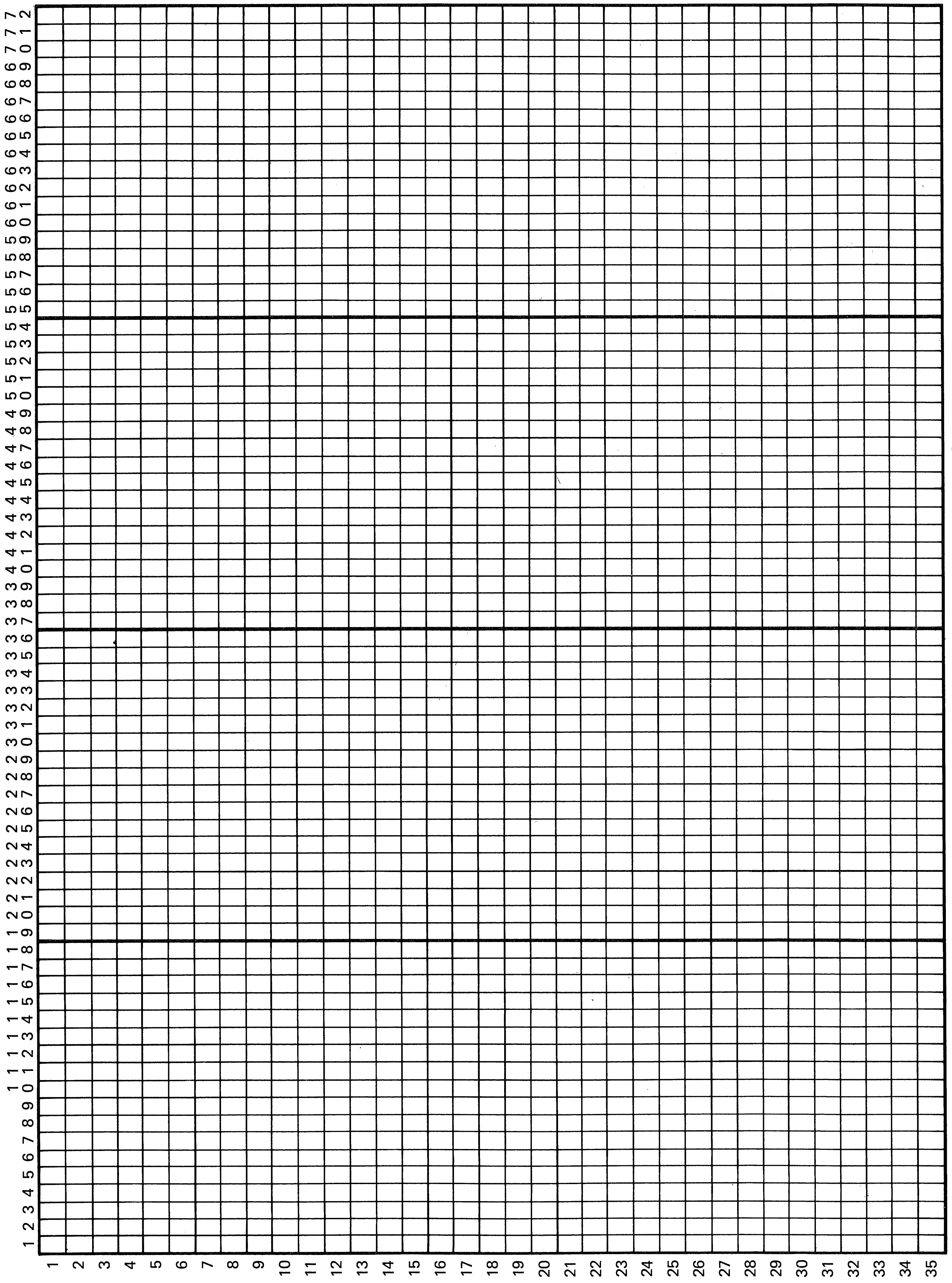
Message Number	Error Message
83	A number specified in the PRINT USING statement contains an exponent outside the range ± 127. Example: 100 IMAGE FD.3D 110 PRINT USING 100:8.5E+200
84	The IMAGE format string was deleted during the PAGE FULL interrupt routine.
85	A portion of the IMAGE format string was 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: 100 IMAGE 5A 110 PRINT USING 100: "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 100.
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.

This work sheet should be useful in designing the organization of formatted output, especially when PRINT USING statements are involved. The vertical axis is labeled 1 through 35, corresponding to the number of lines that the display can contain. The horizontal axis, numbered 1 through 72, corresponds to the number of characters that a line can contain. The four 18 character wide columns correspond to the four print fields that result from the automatic tab operator (i.e., the comma, as in PRINT A,B).

FORMATTING WORKSHEET

PROGRAM TITLE ______

PAGE ____ OF ____



Permission is granted to reproduce this work sheet for customer's formatting purposes only. Reproduction for commercial purposes is expressly forbidden.

INDEX