Any and all material contained herein is supplied without representation or warranty of any kind. The General Electric Company therefore assumes no responsibility and shall have no liability of any kind arising from the supply or use of this publication or any material contained herein.
Preface

This manual, which combines and supersedes two other manuals--BASIC Language Reference Manual (202026A) and BASIC Language Extensions Reference Manual (802207)--describes the version of the BASIC language used with the General Electric Mark I Time-Sharing Service.

Another manual, Mark I Time-Sharing Service Command System Reference Manual (229116) explains all of the system commands that are a part of the Mark I Time-Sharing Service. It should be consulted for system information.

The original development of the BASIC language was supported by the National Science Foundation under the terms of a grant to Dartmouth College. Under this grant, Dartmouth College developed, under the direction of Professors John G. Kemeny and Thomas E. Kurtz, the BASIC language compiler. Since that development, BASIC has been offered as part of the Time-Sharing Service of General Electric's Information Service Department.

General Electric has continued to expand the capabilities of BASIC, adding such versatile features as string manipulation, data files, formatted line output, and others.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION.</td>
<td>1</td>
</tr>
<tr>
<td>1. A BASIC PRIMER</td>
<td>2</td>
</tr>
<tr>
<td>An Example.</td>
<td>2</td>
</tr>
<tr>
<td>Formulas</td>
<td>5</td>
</tr>
<tr>
<td>Numbers</td>
<td>6</td>
</tr>
<tr>
<td>Variables</td>
<td>6</td>
</tr>
<tr>
<td>Loops</td>
<td>7</td>
</tr>
<tr>
<td>Lists and Tables</td>
<td>9</td>
</tr>
<tr>
<td>Errors and Debugging</td>
<td>11</td>
</tr>
<tr>
<td>Summary of Elementary BASIC Statements</td>
<td>14</td>
</tr>
<tr>
<td>LET</td>
<td>14</td>
</tr>
<tr>
<td>READ and DATA</td>
<td>14</td>
</tr>
<tr>
<td>INPUT</td>
<td>15</td>
</tr>
<tr>
<td>PRINT</td>
<td>15</td>
</tr>
<tr>
<td>GØ ÒØ</td>
<td>16</td>
</tr>
<tr>
<td>ØØ--GØ ÒØ</td>
<td>16</td>
</tr>
<tr>
<td>IF--THEN</td>
<td>17</td>
</tr>
<tr>
<td>FOR and NEXT</td>
<td>17</td>
</tr>
<tr>
<td>DIM</td>
<td>18</td>
</tr>
<tr>
<td>END</td>
<td>18</td>
</tr>
<tr>
<td>2. ADVANCED BASIC</td>
<td>19</td>
</tr>
<tr>
<td>Alphanumeric Data and String Manipulation</td>
<td>19</td>
</tr>
<tr>
<td>DIM</td>
<td>19</td>
</tr>
<tr>
<td>LET</td>
<td>19</td>
</tr>
<tr>
<td>READ and DATA</td>
<td>20</td>
</tr>
<tr>
<td>INPUT</td>
<td>20</td>
</tr>
<tr>
<td>PRINT</td>
<td>21</td>
</tr>
<tr>
<td>IF--THEN</td>
<td>21</td>
</tr>
<tr>
<td>More About Printing</td>
<td>21</td>
</tr>
<tr>
<td>PRINT</td>
<td>21</td>
</tr>
<tr>
<td>TAB</td>
<td>23</td>
</tr>
<tr>
<td>Rules for Printing Numbers</td>
<td>23</td>
</tr>
<tr>
<td>PRINT USING and Image Statements</td>
<td>24</td>
</tr>
<tr>
<td>Integer Fields</td>
<td>25</td>
</tr>
<tr>
<td>Decimal Fields</td>
<td>25</td>
</tr>
<tr>
<td>Exponential Fields</td>
<td>26</td>
</tr>
<tr>
<td>Alphanumeric Fields</td>
<td>26</td>
</tr>
<tr>
<td>Literal Fields</td>
<td>27</td>
</tr>
<tr>
<td>General Rules</td>
<td>27</td>
</tr>
<tr>
<td>Functions</td>
<td>28</td>
</tr>
<tr>
<td>INT</td>
<td>28</td>
</tr>
<tr>
<td>RND</td>
<td>28</td>
</tr>
<tr>
<td>SGN</td>
<td>30</td>
</tr>
<tr>
<td>CLK</td>
<td>31</td>
</tr>
<tr>
<td>TIM</td>
<td>31</td>
</tr>
<tr>
<td>DEF</td>
<td>31</td>
</tr>
<tr>
<td>Additional BASIC Statements</td>
<td>31</td>
</tr>
<tr>
<td>GØSUB and RETURN</td>
<td>32</td>
</tr>
</tbody>
</table>
## CONTENTS (CONT'D)

### 2. ADVANCED BASIC STATEMENTS (Cont'd)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>33</td>
</tr>
<tr>
<td>STOP</td>
<td>35</td>
</tr>
<tr>
<td>REM</td>
<td>35</td>
</tr>
<tr>
<td>RESTORE</td>
<td>36</td>
</tr>
<tr>
<td>CHAIN</td>
<td>36</td>
</tr>
<tr>
<td>Data Files</td>
<td>37</td>
</tr>
<tr>
<td>File Reference</td>
<td>37</td>
</tr>
<tr>
<td>File Designator</td>
<td>38</td>
</tr>
<tr>
<td>Dummy Catalog Files</td>
<td>38</td>
</tr>
<tr>
<td>BCD Data Files</td>
<td>39</td>
</tr>
<tr>
<td>File Reading</td>
<td>39</td>
</tr>
<tr>
<td>File Writing</td>
<td>40</td>
</tr>
<tr>
<td>End-of-File and End-of-Space</td>
<td>41</td>
</tr>
<tr>
<td>End-of-File Test</td>
<td>41</td>
</tr>
<tr>
<td>File Restoring</td>
<td>42</td>
</tr>
<tr>
<td>File Scratching</td>
<td>42</td>
</tr>
<tr>
<td>File Backspacing</td>
<td>43</td>
</tr>
<tr>
<td>Binary Data Files</td>
<td>43</td>
</tr>
<tr>
<td>File Reading</td>
<td>43</td>
</tr>
<tr>
<td>File Writing</td>
<td>45</td>
</tr>
<tr>
<td>Random Accessing</td>
<td>45</td>
</tr>
<tr>
<td>Locating the Element Pointer</td>
<td>46</td>
</tr>
<tr>
<td>File Scratching</td>
<td>46</td>
</tr>
<tr>
<td>File Restoring</td>
<td>47</td>
</tr>
<tr>
<td>End-of-File Test</td>
<td>47</td>
</tr>
<tr>
<td>Matrices</td>
<td>48</td>
</tr>
<tr>
<td>MAT READ and MAT PRINT</td>
<td>49</td>
</tr>
<tr>
<td>Matrix Addition, Subtraction, and Multiplication</td>
<td>49</td>
</tr>
<tr>
<td>Scalar Multiplication</td>
<td>50</td>
</tr>
<tr>
<td>Identity Matrix</td>
<td>50</td>
</tr>
<tr>
<td>Matrix Transposition</td>
<td>50</td>
</tr>
<tr>
<td>Matrix Inversion</td>
<td>51</td>
</tr>
<tr>
<td>Matrix ZER and CON Functions</td>
<td>51</td>
</tr>
<tr>
<td>Dimensioning</td>
<td>51</td>
</tr>
<tr>
<td>Examples</td>
<td>52</td>
</tr>
</tbody>
</table>

### A. ERROR MESSAGES

### B. LIMITATIONS ON BASIC

### C. COMPARISON ORDER FOR BASIC CHARACTERS

### D. USING THE TIME-SHARING SYSTEM

### INDEX

iv
Introduction

A program is a set of directions that is used to tell a computer how to provide an answer to some problem. It usually starts with the given data, contains a set of instructions to be carried out in a certain order, and ends up with a set of answers.

Any program must meet two requirements before it can be carried out. The first is that it must be presented in a form that the computer understands. If the program is a set of instructions for solving a system of linear equations and the computer is an English-speaking person, the program will be presented in a combination of mathematical notation and English. If the computer is a French-speaking person, the program will be presented in French rather than English. And if the computer is a high-speed digital computer, the program must be presented in a programming language that the digital computer understands.

The second requirement for any program is that it must be completely and precisely stated. This requirement is crucial when dealing with a digital computer, which has no ability to infer what you mean. It does what you tell it to do, not what you meant to tell it.

We are talking about programs that provide numerical answers to numerical problems. It would be easy to present a program in the English language, but such a program would pose insurmountable difficulties for the computer. English is rich with ambiguities and redundancies, the qualities that make poetry possible but computing impossible. Instead of using English, you present your program in a programming language that resembles ordinary mathematical notation, that has a simple vocabulary, and that permits a complete and precise specification of your program. The programming language you will use is BASIC, Beginner's All-purpose Symbolic Instruction Code. BASIC is precise, simple, and easy to understand.

An introduction to writing a BASIC program is given in Chapter 1, which includes all that you need to know to write a variety of useful and interesting programs. Chapter 2 deals with more advanced techniques. The Appendixes contain a variety of reference materials.
1. A BASIC Primer

AN EXAMPLE

The following example is a complete BASIC program for solving a system of two simultaneous linear equations in two variables:

\[ \begin{align*} ax + by &= c \\ dx + ey &= f \end{align*} \]

and then solving two systems, each differing from this system only in the constants \( c \) and \( f \).

If \( ae - bd \) is not equal to zero, you should be able to solve this system to find that

\[ x = \frac{ce - bf}{ae - bd} \quad \text{and} \quad y = \frac{af - cd}{ae - bd} \]

If \( ae - bd \) is equal to zero, either there is no solution or there are infinitely many, but there is no unique solution.

If you are rusty on solving such systems, take our word for it that this is correct. For now, we simply want you to understand the BASIC program for solving this system.

Study the following program carefully--the purpose of most lines in the program is self-evident--and then read the commentary and explanation.

```
100 READ A,B,D,E
110 LET G=A*E-B*D
120 IF G=0 THEN 180
130 READ C,F
140 LET X=(C*E-B*F)/G
150 LET Y=(A*F-C*D)/G
160 PRINT X,Y
170 G0 TO 130
180 PRINT "NO UNIQUE SOLUTION"
190 DATA 1,2,4
200 DATA 2,-7,5
210 DATA 1,3,4,-7
999 END
```

You can see, first, that this sample program uses only capital letters, because the teletype-writer has only capital letters.

Second, you can see that each line of the program begins with a line number. These numbers are called line numbers; they identify the lines, each of which is called a statement. A program is made up of statements, most of which are instructions to the computer. The line numbers also serve to specify the order in which the statements are to be performed by the computer. This means that you may type your program in any order. Before the program is run, the computer sorts out and edits the program, putting the statements into the order specified by their line numbers. This editing process also facilitates correcting and changing programs, as we shall explain later on.

Third, note that each statement starts, after its line number, with an English word. The word denotes the type of the statement. There are several types of statements in BASIC, nine
of which are discussed in this chapter. Seven of these nine appear in the sample program we are now considering.

Note also that, although it is not obvious from the program, spaces have no significance in BASIC statements, except in messages that are to be printed out, as in line number 180. Spaces may be used or not, at will, to make a program more readable. Statement 100 could have been typed as 100READ A, B, D, E and statement 110 as 110 LET G = A * E - B * D.

With this preface, let's go through the program, step by step. The first statement, 100, is a READ statement. It must be accompanied by one or more DATA statements. When the computer encounters a READ statement while executing your program, it will cause the variables listed after the word READ to be given values according to the next available numbers in the DATA statements. In our sample program, we read A in statement 100 and assign the value 1 to it from statement 190, and similarly with B and 2, and with D and 4. At this point, we have exhausted the available data in statement 190, but there is more in statement 200. We pick up there the number 2 to be assigned to E.

Next we go to statement 110, a LET statement, where we first encounter a formula to be evaluated. (The asterisk, *, is used to denote multiplication.) In this statement we direct the computer to find the value of AE - BD, and to call the result G. In general, a LET statement directs the computer to set a variable equal to the formula on the righthand side of the equal sign.

We know that if G is equal to zero, the system has no unique solution. Therefore, we next ask, in line 120, whether G is equal to zero. If the computer finds a Yes answer to the question, it is directed to go to line 180. Line 180 tells it to print out NO UNIQUE SOLUTION. From this point, it would go to the next statement. But lines 190, 200, and 210 give it no instructions, since DATA statements are not executed, and it then goes to line 999, which directs it to END the program.

If the answer to the question "Is G equal to zero?" is No, as it is in our sample program, the computer simply goes to the next statement, in this case statement 130. (An IF--THEN statement tells the computer where to go if the IF condition exists, but to go on to the next statement if it does not exist.) Statement 130 directs the computer to read the next two entries from the DATA statements--in this case -7 and 5, both in statement 200--and to assign them to C and F respectively. The computer is now ready to solve the system:

\[
\begin{align*}
x + 2y &= -7 \\
4x + 2y &= 5
\end{align*}
\]

In statements 140 and 150, we direct the computer to find the values of x and y according to the formulas provided. Note that we must use parentheses to show that CE - BF is divided by G. Without parentheses, only BF would be divided by G, and the computer would find

\[
X = \frac{CE - BF}{G}
\]

The computer is told in line 160 to print the two values computed, those of X and Y. Having done so, it moves on to line 170, where it is directed back to line 130. If there are additional numbers in the DATA statements, as there are here in 210, the computer is told in line 130 to take the next number and assign it to C, and the one after that to F. The computer is now ready to solve the system:

\[
\begin{align*}
x + 2y &= 1 \\
4x + 2y &= 3
\end{align*}
\]

As before, it finds the solution in 140 and 150, prints out the values in 160, and then is directed in 170 to go back to 130.

In line 130 the computer reads two more values, 4 and -7, which it finds in line 210. It then solves the system:

\[
\begin{align*}
x + 2y &= 4 \\
4x + 2y &= -7
\end{align*}
\]
and prints out the solutions. It is directed back again to 130, but there are no more pairs of numbers available for C and F in the DATA statements. The computer therefore informs you that it is out of data, printing on the paper in the teletypewriter OUT OF DATA IN 130, and stops.

Let's look at the importance of the various statements. For example, what would have happened if we had omitted line number 160? The answer is simple. The computer would have solved the equations three times and then told us it was out of data. However, since it was not told to show us (PRINT) the answers, it would not do so, and the solutions would be the computer's secret.

What would have happened if we had left out line 120? In the problem just solved, nothing. But if G were equal to zero, we would have set the computer the impossible task of dividing by zero in 140 and 150, and it would tell us so by printing out DIVISION BY ZERO IN 140 and DIVISION BY ZERO IN 150. Suppose we had left out statement 170? The computer would have solved the first system, printed out the values of X and Y, and then gone on to line 180. As directed, it would print out NO UNIQUE SOLUTION, and then stop.

A natural question that may arise is, why this selection of line numbers? The answer is that the particular choice of line numbers is arbitrary. The only requirement is that statements be numbered in the order that we want the computer to follow in executing the program. We could have numbered the statements 1, 2, 3, 4, ..., 13; but we do not recommend this numbering. We would normally number the statements 100, 110, 120, ..., 999. We put the numbers such a distance apart so that we can later insert additional statements easily. For example, if we find that we have left out two statements belonging between those numbered 140 and 150, we can give them any two numbers between 140 and 150, say 144 and 146. In the editing and sorting process, the computer will put them in the correct place.

Another question that may arise has to do with the placing of the elements of data in the DATA statements: Why place them as they are in the sample program? The choice is arbitrary. We need only arrange the numbers in the order that we want them read—the first for A, the second for B, the third for D, the fourth for E, the fifth for C, the sixth for F, the seventh for the next C, and so on. In place of the three statements numbered 190, 200, 210, we could have put

195 DATA 1, 2, 4, 2, -7, 5, 1, 3, 4, -7

or we could have written, perhaps more logically

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

putting the coefficients in the first DATA statement and the three pairs of righthand constants in the following DATA statements.

After typing the program, we type RUN followed by a carriage return. Up to this point the computer stores the program and does nothing else with it. It is the command RUN that directs the computer to execute the program.

The sample program and the resulting printout are shown now as they appear on the teletypewriter.

```
100 READ A,B,D,E
110 LET G=A*E-B*D
120 IF G=0 THEN 180
130 READ C,F
140 LET X=(C*E-B*F)/G
150 LET Y=(A*F-C*D)/G
160 PRINT X,Y
170 GOTO 130
```
The computer can carry out a great many operations. It can add, subtract, multiply, divide, extract square roots, raise a number to a power, find the sine of a number on an angle measured in radians, and so on. We shall now learn how to tell the computer to carry out these various operations in the order that we want them done.

The computer computes by evaluating formulas that are supplied in a program. The formulas are similar to those used in ordinary mathematical calculation, except that each BASIC formula must be written on a single line. Five arithmetic operations can be used to write a formula. They are listed in the following table.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>A + B</td>
<td>Addition. Add B to A.</td>
</tr>
<tr>
<td>-</td>
<td>A - B</td>
<td>Subtraction. Subtract B from A.</td>
</tr>
<tr>
<td>*</td>
<td>A * B</td>
<td>Multiplication. Multiply B by A.</td>
</tr>
<tr>
<td>/</td>
<td>A / B</td>
<td>Division. Divide A by B.</td>
</tr>
<tr>
<td>^</td>
<td>X ^ 2</td>
<td>Raise to the power. Find X².</td>
</tr>
</tbody>
</table>

We must be careful with parentheses to make sure that we group together the things we want together. We must also understand the order in which the computer does its work.

For example, if we type A+B*C^D, the computer will first raise C to the power D, then multiply this result by B, and then add A to the resulting product. This is the usual convention for A + BC^D. If this is not the intended order, we must use parentheses to indicate a different order. Suppose it is the product of B and C that we want raised to the power D. We must write A + (B*C)^D. Or, if we want to multiply A + B by C to the power D, we write (A+B)*C^D. We could add A to B, multiply their sum by C, and raise the product to the power D by writing ((A+B)*C)^D.

The order of priorities for computing is according to the following rules.

1. The formula inside parentheses is computed before the enclosed quantity is used in further calculations.

2. In the absence of parentheses in a formula that includes addition, multiplication, and the raising of a number to a power, the computer first raises the number to the power, then does the multiplication, and does the addition last. Division has the same priority as multiplication, and subtraction the same as addition.

3. In the absence of parentheses in a formula that includes only multiplication and division (or only addition and subtraction), the computer works from left to right.

The rules are illustrated in the sample program already considered. The rules also tell us that the computer, given A-B-C, will subtract B from A and then C from their difference. Given A/B/C, it will divide A by B and then that quotient by C. Given A\(B/C\), the computer
will raise the number A to the power B and then raise the resulting number to the power C. If there is ever any question in your mind about the priority, put in more parentheses to avoid possible ambiguities.

In addition to the five arithmetic operations, the computer can evaluate several mathematical functions. The functions are given special three-letter names, as shown in the following table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIN(X)</td>
<td>Find the sine of X.</td>
</tr>
<tr>
<td>COS(X)</td>
<td>Find the cosine of X.</td>
</tr>
<tr>
<td>TAN(X)</td>
<td>Find the tangent of X.</td>
</tr>
<tr>
<td>ATN(X)</td>
<td>Find the arctangent of X.</td>
</tr>
<tr>
<td>EXP(X)</td>
<td>Find e^x.</td>
</tr>
<tr>
<td>LOG(X)</td>
<td>Find the natural logarithm of X (ln X).</td>
</tr>
<tr>
<td>ABS(X)</td>
<td>Find the absolute value of X (</td>
</tr>
<tr>
<td>SQR(X)</td>
<td>Find the square root of X (√X).</td>
</tr>
</tbody>
</table>

Three other mathematical functions are available in BASIC: INT, RND, and SGN. These are reserved for explanation in Chapter 2.

In place of X, we may substitute any formula or number in parentheses following any of the functions listed above. For example, we may tell the computer to find $\sqrt{4 + X^2}$ by writing SQR(4+X:3), or the arctangent of $3X - 2e^x + 8$ by writing ATN(3*X-2*EXP(X)+8).

Since we have mentioned numbers and variables, we should be sure we understand how to write numbers for the computer and what variables are allowed.

Numbers

A number may be positive or negative, and it may contain as many as nine digits, but it must be expressed in decimal form. For example, all of the following are numbers in BASIC:

2  -3.675  123456789  -.987654321  483.4156

The following are not numbers in BASIC:

14/3  √7  .00123456789

The first two are formulas, but not numbers. The last one has more than nine digits. We may tell the computer to find the decimal expansion of $14/3$ or $\sqrt{7}$, and to do something with the resulting number, but we may not include either in a list of DATA.

We gain flexibility by use of the letter E, which stands for "times ten to the power." Using E, we can write .00123456789 in several forms acceptable to the computer: .123456789E-2 or 123456789E-11 or 1234.56789E-6. We can write ten million as 1E7 and 1969 as 1.969E3. We do not write a number as E7, but must write 1E7 to indicate that it is 1 that is multiplied by 10^7.

Variables

A variable in BASIC is denoted by any letter, or by any letter followed by a single digit. The computer will interpret E7 as a variable, along with A, X, N5, IO, and $\ddot{O}$. A variable in BASIC stands for a number, usually one that is not known to the programmer at the time the program is written. Variables are given or assigned values by LET, READ, and INPUT statements. All variables have the initial value of zero, so that if you want the starting value of a variable to be zero you need not assign it that value. (Another kind of variable, the string variable, is discussed later on in Chapter 2.)
Although the computer does little in the way of correcting during computation, it will sometimes help you when you forget to indicate absolute value. For example, if you ask for the square root of -7 or the logarithm of -5, the computer will give you the square root of 7 with the error message that you have asked for the square root of a negative number, or the logarithm of 5 with the error message that you have asked for the logarithm of a negative number.

Three other mathematical symbols, symbols of relation, are available in BASIC to indicate any of six standard relations. These are used in IF--THEN statements, where values must be compared. The six possible relations are shown in the following table.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>A = B</td>
<td>Is equal to. A is equal to B.</td>
</tr>
<tr>
<td>&lt;</td>
<td>A &lt; B</td>
<td>Is less than. A is less than B.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>A &lt;= B</td>
<td>Is less than or equal to. A is less than or equal to B.</td>
</tr>
<tr>
<td>&gt;</td>
<td>A &gt; B</td>
<td>Is greater than. A is greater than B.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>A &gt;= B</td>
<td>Is greater than or equal to. A is greater than or equal to B.</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>A &lt;&gt; B</td>
<td>Is not equal to. A is not equal to B.</td>
</tr>
</tbody>
</table>

**LOOPS**

We are often interested in writing a program in which one or more parts are traversed not just once, but a number of times, perhaps with slight changes each time. In order to write the simplest program, one in which the part to be repeated is written just once, we use the programming device known as a loop.

The use of loops can be illustrated by two programs for the simple task of printing out a table of the first 100 positive integers together with the square root of each. Without a loop, our program would be 101 lines long:

```
100 PRINT I,SQR(I)
105 PRINT 2,SQR(2)
110 PRINT 3,SQR(3)
···
590 PRINT 99,SQR(99)
595 PRINT 100,SQR(100)
600 END
```

With the following program, using one type of loop, we can get the same table with only 5 lines of instruction instead of 101.

```
100 LET X=1
110 PRINT X,SQR(X)
120 LET X=X+1
130 IF X<=100 THEN 110
999 END
```

Statement 100 gives the value of 1 to X, which initializes the loop. Line 110 causes the printing of both 1 and its square root. Line 120 increases the value of X by 1, to 2. Line 130 asks whether X is less than or equal to 100--a Yes answer directs the computer back to line 110. Here it prints 2 and \(\sqrt{2}\), and goes to 120. Again X is increased by 1, this time to 3, and at 130 it goes back to 110. This process is repeated--line 110 (print 3 and \(\sqrt{3}\)), line 120 (X = 4), line 130 (since 4 is less than or equal to 100 go back to line 110), and so on--until the loop has been traversed 100 times. Then X becomes 101. The computer now finds a No answer to the question in line 130 (X is greater than 100, not less than or equal to 100). It therefore does not return to 110 but moves on to line 999, and ends the program.

All loops contain four elements: initialization (line 100 in our program), the body (line 110), modification (line 120), and an exit test (line 130). Because loops are so important, BASIC...
provides a pair of statements that specify a loop even more simply than the previous program. They are the FOR and NEXT statements. Their use is illustrated in this program:

```plaintext
100 FOR X=1 TO 100
110 PRINT X, SQRT(X)
120 NEXT X
999 END
```

which does exactly the same thing as the two previous programs. In line 100, X is set equal to 1, and a test is set up, like that of line 130 above. Line 120 causes X to be increased by 1, and also carries out the test to decide whether to go back to line 110 or to go on. Thus lines 100 and 120 take place of lines 100, 120, and 130 in the previous program—and they are easier to use.

Note that the value of X is increased by 1 each time we go through the loop. If we wanted a different increase, we could specify it by writing, for example:

```plaintext
100 FOR X=1 TO 100 STEP 5
```

and the computer would assign 1 to X on the first time through the loop, 6 to X on the second time through, 11 on the third time, and 96 on the twentieth time. Another step of 5 would take X beyond 100, so the program would go on to the end after printing 96 and its square root. The step value may be either positive or negative. We could obtain the first table printed in reverse order by writing line 100 as

```plaintext
100 FOR X=100 TO 1 STEP -1
```

In the absence of a STEP instruction, a step size of +1 is assumed.

More complicated FOR statements are allowed. The initial value, the final value, and the step size may all be formulas of any complexity. For example, if N and Z have been specified earlier in the program, we could write

```plaintext
250 FOR X=N+7*Z TO (Z-N)/3 STEP (N-4*Z)/10
```

For a positive step size, the loop continues as long as the control variable is less than or equal to the final value. For a negative step size, the loop continues as long as the control variable is greater than or equal to the final value.

If the initial value is greater than the final value (or less than for a negative step size), then the body of the loop will not be done even once. The computer will immediately pass to the statement following the NEXT. As an example, the following program for adding up the first n integers will give the correct result 0 when n is 0.

```plaintext
100 READ N
110 FOR K=1 TO N
120 LET S=S+K
130 NEXT K
140 PRINT S
150 G0 T0 100
160 DATA 3,10,0
999 END
```

It is often useful to have loops within loops. These are called nested loops. They can be expressed with FOR and NEXT statements. But they must actually be nested and must not cross, as the following skeleton examples illustrate.
In addition to the ordinary numeric variables used in BASIC, there are variables that we can use to designate the elements of a list or table. We use these where we would ordinarily use a subscript or a double subscript, as for the coefficients of a polynomial \((a_0, a_1, a_2, \ldots)\) or the elements of a matrix \((b_{i,j})\). The variables that we use in BASIC consist of a single letter, which we call the name of the list or table, followed by the subscripts in parentheses. For the coefficients of the polynomial we would write \(A(0), A(1), A(2), \ldots\); for the elements of the matrix we would write \(B(1,1), B(1,2), \ldots\).

We can enter the list \(A(0), A(1), \ldots A(10)\) into a program very simply with four lines:

```
100 FOR I=0 TO 10
110 READ A(I)
120 NEXT I
130 DATA 2,3,-5,7,2,4,-9,123,4,-4,3
```

We need no special instruction to the computer if no subscript greater than 10 occurs. If we want larger subscripts, we must use a dimension (DIM) statement, to tell the computer to save extra space for the list or table. When in doubt, indicate a larger dimension than you expect to use. For example, if we want a list of 15 numbers entered, we might write:

```
100 DIM A(25)
110 READ N
120 FOR I=1 TO N
130 READ A(I)
140 NEXT I
150 DATA 15
160 DATA 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
```

Statements 110 and 150 could have been eliminated by writing 120 as \(FOR I=1 TO 15\), but the form we used allows us to lengthen the list by changing only statement 150, so long as the number of elements in the list does not exceed 25.

We would enter a \(3 \times 5\) table into a program by writing:

```
100 FOR I=1 TO 3
110 FOR J=1 TO 5
120 READ B(I,J)
130 NEXT J
140 NEXT I
150 DATA 2,3,-5,-9,2
160 DATA 4,-7,3,4,-2
170 DATA 3,-3,5,7,8
```

We may enter a table with no dimension statement, and the computer will handle all the entries from \(B(0,0)\) to \(B(10,10)\). But if you try to enter a table with a subscript greater than 10, without a DIM statement, you will get an error message telling you that you have a subscript error. This can be easily corrected by entering, for example, the line:
which will reserve space for a 20 by 30 table.

The single letter denoting a list or table name may also be used to denote a simple variable without confusion. But the same letter may not be used to denote both a list and a table in the same program.

The form of the subscript is quite flexible. You might have the list item B(I+K) or the table items B(I,K) or Q(A(3,7),B-C).

Let's look now at a sample program that uses both a list and a table. The program computes the total sales of each of five salesmen, each of whom sells the same three products.

```
95 DIM B(20,30)

SALES I 14:38
100 FOR I=1 TO 3
110 READ P(I)
120 NEXT I

130 FOR J=1 TO 5
140 FOR I=1 TO 5
150 READ S(I,J)
160 NEXT J
170 NEXT I
180 FOR J=1 TO 5
190 LET S=0
200 FOR I=1 TO 3
210 LET S=S+P(I)*S(I,J)
220 NEXT I
230 PRINT "TOTAL SALES FOR SALESMAN " JJ,"$";S
240 NEXT J
250 DATA 1.25,4.30,2.50
260 DATA 40,20,37,29,42
270 DATA 10,16,3,21,8
280 DATA 35,47,29,16,33
999 END

RUN

SALES I 14:39
TOTAL SALES FOR SALESMAN 1 $ 180.5
TOTAL SALES FOR SALESMAN 2 $ 211.3
TOTAL SALES FOR SALESMAN 3 $ 131.65
TOTAL SALES FOR SALESMAN 4 $ 166.55
TOTAL SALES FOR SALESMAN 5 $ 169.4
```

The list P gives the price per item of each of the three products. The table S tells how many items of each product each man sold. As you can see from the program, product number 1 sells for $1.25 per item, number 2 for $4.30 per item, and number 3 for $2.50 per item. You can see also that salesman number 1 sold 40 items of the first product, 10 of the second, and 35 of the third, and so on. The program reads in the price list in lines 100, 110, and 120, using data in line 250. It reads in the sales table in lines 130-170, using data in lines 260-280. The same program could be used again, modifying only line 250 if the prices change, and only lines 260-280 to enter the sales in another month.

The sample program did not need a dimension statement, since the computer automatically saves enough space to allow subscripts to run from 0 to 10. A DIM statement is normally used to save more space. But in a long program, requiring many small tables, DIM may be used to save less space for tables, in order to leave more for the program.

Since a DIM statement is not executed, it may be entered into the program on any line before END. It is convenient, though, to place DIM statements near the beginning of the program.
ERRORS AND DEBUGGING

Occasionally the first run of a new problem will be free of errors and give the correct answers. Usually, though, errors will have to be corrected before the program runs right. Errors are of two types: errors of form that prevent the running of the program, and logical errors in the program that cause the computer to produce either wrong answers or no answers at all.

Errors of form will cause error messages to be printed. The various error messages are listed and explained in Appendix A. Logical errors are often much harder to find, particularly when the program gives answers that are nearly correct. In either case, after you find the errors, you can correct them by changing lines, inserting new lines, or deleting lines from the program. A line is changed by typing it correctly with the same line number. A line is inserted by typing it with a line number between those of two existing lines. A line is deleted by typing its line number and pressing the RETURN key. Notice that you can insert a line only if the original line numbers are not consecutive. For this reason, most programmers start out using line numbers that are multiples of five or ten, but that is a matter of choice.

You can make corrections at any time that you notice them, either before or after a run. Since the computer sorts lines and arranges them in order, a line may be retyped out of sequence. Simply retype the bad line with its original line number.

As with most problems in computing, we can best illustrate the process of finding errors (or bugs) in a program, and correcting (or debugging) it, by an example. Let's consider the problem of finding the value of X between 0 and 3 for which the sine of X is a maximum, and printing out this value of X and the value of its sine.

If you have studied trigonometry, you know that \( \frac{\pi}{2} \) is the correct value of X, but we shall use the computer to test successive values of X from 0 to 3. First we shall use intervals of .1, then of .01, and finally of .001.

Thus, we shall tell the computer to find the sine of 0, of .1, of .2, of .3, ..., of 2.8, of 2.9, and of 3, and to determine which of these 31 values is the largest. It will do so by testing \( \text{SIN}(0) \) and \( \text{SIN}(.1) \) to see which is larger, and calling the larger of these two numbers \( M \). Then it will pick the larger of \( M \) and \( \text{SIN}(.2) \), and call it \( M \). This number it will check against \( \text{SIN}(.3) \), and so on down the line. Each time a larger value of \( M \) is found, the value of \( X \) is remembered in \( X_0 \). When the computer finished the series, \( M \) will have the value of the largest of the 31 sines, and \( X_0 \) will be the argument that produced that largest value. It will then repeat the search, this time checking the 301 numbers 0, .01, .02, .03, ..., 2.98, 2.99, and 3, finding the sine of each and checking to see which sine is the largest. Finally, it will check the 3001 numbers 0, .001, .002, .003, ..., 2.998, 2.999, and 3, to find which has the largest sine. At the end of each of the three searches, we want the computer to print three numbers: the value \( X_0 \) that has the largest sine, the sine of that number, and the interval of search.

Before going to the terminal, we write a program. Let's assume it is the following:

```plaintext
100 READ D
110 LET X0=0
120 FOR X=0 TO 3 STEP D
130 IF SIN(X) =M THEN 190
140 LET X0=X
150 LET M=SIN(X0)
160 PRINT X0,X,D
170 NEXT X
180 GO TO 110
190 DATA .1,.01,.001
999 END
```

We shall illustrate the entire sequence on the teletypewriter, and make explanatory comments on the righthand side.
NEW
NEW FILE NAME--MAXSIN
READY.

100 READ D
110 LET XO=0
120 FOR X=0 TO 3 STEP D
130 IF SIN(X)<=M THEN 190
140 LET XO=X
150 LET M=SIN(X)
160 PRINT XO,X,D
170 NEXT X0
180 G0 T0 110
110 LET XO=0
190 DATA .1,.01,.001
999 END
RUN
WAIT.

MAXSIN 14:42

INCORRECT FORMAT IN 160
NEXT WITHOUT FOR IN 170
FOR WITHOUT NEXT
UNDEFINED NUMBER 190

USED 0.83 UNITS.
160 PRINT XO,X,D
170 NEXT X
130 IF SIN(X)<=M THEN 170
RUN

This is obviously incorrect. Every value of X is being printed. We stop the printing by pressing the BREAK key. Then we ponder the program for a while, trying to figure out what's wrong with it.

We notice that SIN(0) is compared with M on the first time through the loop, but we had assigned no value to M. So we wonder if giving a value less than the maximum value of the sine will do it. We give it the value of -1, by changing line 110, where we had incorrectly initialized X0 instead of M.

We are about to print out almost the same table as before. It is printing out X0, the current value of X, and the interval size each time it goes through the loop.

We fix this by moving the PRINT statement outside the loop. We delete line 160, and line 175 is outside of the loop. We also realize that we want M printed and not X.

After typing line 180, we notice that LET was mistyped in line 110, so we retype it, this time correctly.

When we receive the first error message, we inspect line 160 and find that we used X0 instead of X0 for a variable. The next two error messages refer to lines 120 and 170, where we see that we mixed variables. We correct this by changing line 170. The fourth error message points out that line 130 directed the computer to a DATA statement and not to line 170 where it should go. We correct this by retyping line 130.
We see that we are doing the same thing over and over again, the case for D=.1. So we stop the printing and inspect the program again.

Of course. Line 180 sent us back to line 110 to repeat the operation rather than back to line 100 to pick up a new value for D. We also decide to put in headings for our columns by a PRINT statement.

There is an error in our PRINT statement: no lefthand quotation mark for the third item.

We retype line 95 with all of the required quotation marks.

Exactly the desired results. Of the 31 numbers 0, .1, .2, .3, ..., 2.8, 2.9, 3, it is 1.6 that has the largest sine, namely .999574. Similarly for the finer subdivisions.

The whole process used only 18.33 computer resource units.

Having changed so many parts of the program, we ask for a list of the corrected program.

We save the program for later use. This should not be done unless we do expect to use the program later.
In solving this problem, there are two common devices we did not use. One is the insertion of a PRINT statement when we wonder whether the machine is computing what we think we asked it to compute. For example, if we wondered about $M$, we could have inserted 155 PRINT $M$, and we would have seen the values. The other device is used after several corrections have been made and you are not quite sure what the program now looks like. Simply type LIST or LISTNH, and the computer will type out the program in its current form for you to inspect.

**SUMMARY OF ELEMENTARY BASIC STATEMENTS**

In this section, we shall give a concise description of each of the types of BASIC statements you will find most useful in writing the simpler kinds of BASIC programs. For each form, we shall assume a line number and use underlining to denote a general type. Thus, variable means any variable, which is a single letter, possibly followed by a single digit.

**LET**

The LET statement is not a statement of algebraic equality. It is an instruction to the computer to do certain computations and to assign the answer to a certain variable. Each LET statement is of the form

\[
\text{LET } \text{variable } = \text{formula}
\]

Examples:

100 LET X=X+1
259 LET W7=(W-X^3)*(Z-A/(A-B))-17

**READ and DATA**

We use a READ statement to assign to the listed variables values obtained from a DATA statement. Neither statement is used without one of the other type. A READ statement causes the variables listed in it to be given, in order, the next available numbers in the collection of DATA statements. Before the program is run, the computer puts all of the DATA statements, in the order in which they appear, into a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data, with a READ statement still asking for more, the program is assumed to be done.

Since we have to read in data before we can work with it, READ statements normally are placed near the beginning of a program. The location of DATA statements is unimportant, so long as they are in the correct order. A common practice is to put all DATA statements together just before the END statement.

Each READ statement is of the form

\[
\text{READ sequence of variables}
\]

and each DATA statement is of the form

\[
\text{DATA sequence of numbers}
\]

Examples:

150 READ X,Y,Z,X1,Y2,Q9
330 DATA 4,2,1.7
340 DATA 6.734E-3,-174.321,3.14159265
234 READ B(K)
263 DATA 2,3,5,7,9,11,10,8,6,4
100 READ R(I,J)
440 DATA -3.5,-9,2,37,2.9876,-437.234E-5
450 DATA 2.765,5.5576,2.3789E2

Remember that only numbers are put in a DATA statement, and that $15/7$ and $\sqrt{3}$ are formulas, not numbers.
**INPUT**

At times it is desirable to have data entered during running of a program. This is particularly true when one person writes the program and saves it in the computer's memory, and other persons are to supply the data. This may be done by using an INPUT statement, which is very much like a READ statement, but does not draw numbers from a DATA statement. Each INPUT statement is of the form

```
INPUT sequence of variables
```

If, for example you want the user to supply values for X and Y in a program, you include the statement

```
140 INPUT X,Y
```

before the first statement that is to use either of the two values. When it encounters the INPUT statement, the computer types a question mark on the printout and waits for input. The user types two numbers, separated by a comma, and presses the return key, and the computer goes on with the rest of the program.

Frequently an INPUT statement is accompanied by a PRINT statement, to make sure that the user knows what the question mark is asking for. If you include in your program

```
120 PRINT "YOUR VALUES OF X, Y, AND Z ARE";
130 INPUT X,Y,Z
```

the computer will type out

```
YOUR VALUES OF X, Y, AND Z ARE?
```

Without the semicolon at the end of line 120, the question mark would have been printed on the next line.

Data entered by INPUT statement is not saved with the program. Also, it may take a long time to enter a large amount of data using INPUT. INPUT should be used only when small amounts of data are to be entered, or when it is necessary to enter data during the program run, as it is with game playing programs.

**PRINT**

The PRINT statement has a number of different uses. It is discussed in more detail in Chapter 2. The common uses are:

A. To print out the result of some computations
B. To print out verbatim a message included in the program
C. To do a combination of A and B
D. To skip a line

We have seen examples of only A and B in our sample programs. Each type is slightly different in form, but all start with PRINT after the line number.

Examples of type A:

```
100 PRINT X,SQR(X)
135 PRINT X,Y,Z,B*B-4*A*C,EXP(A-B)
```

The first will print the value of X and then, a few spaces to the right, its square root. The second will print five different numbers: X, Y, B^2-4AC, and e^{A-B}. The computer will compute the two formulas and print the values for you, if you have already given values to A, B, and C. It can print up to five numbers per line in this format.
Examples of type B:

100 PRINT "NO UNIQUE SOLUTION"
430 PRINT "X VALUE", "SINE", "RESOLUTION"

You have seen both in the sample programs. The first prints the statement within the quotation marks. The second prints the three labels with spaces between them. The labels in 430 automatically line up with three numbers called for in a PRINT statement, as seen in the program MAXSIN.

Examples of type C:

150 PRINT "THE VALUE OF X IS";X
315 PRINT "THE SQUARE ROOT OF";X;"IS";SQR(X)

If the first has computed the value of X as 3, it will print out

THE VALUE OF X IS 3

If the second has computed the value of X as 625, it will print out

THE SQUARE ROOT OF 625 IS 25

In statements of type C, the semicolon is used to minimize space.

Example of type D:

250 PRINT

The computer will advance the paper one line when it encounters this statement.

GØ TØ

There are times in a program when you do not want all statements executed in the order in which they appear in the program. An example of this occurs in the MAXSIN program where the computer has computed X0, M, and D and printed them out in line 160. We did not want the program to go on to the END statement yet, but we wanted it to go through the same process for a different value of D. So we directed the computer to go back to line 100 with a GØ TØ statement. Each GØ TØ statement is of the form

GØ TØ line number

Example:

150 GØ TØ 75

ØN—GØ TØ

The simple GØ TØ statement provides a single branched switch. The ØN--GØ TØ statement provides a multi-branched switch. The form of the statement is

ØN expression GØ TØ line number, line number, ..., line number

The expression is any valid BASIC expression, and the line numbers are those to which the statement will transfer depending on the value of the expression.

Example:

230 ØN X+Y GØ TØ 575, 490, 150
This statement will transfer to line 575, 490, or 150 depending on whether the value of the expression \( X+Y \) is 1, 2, or 3.

The expression value will be truncated to an integer if it is not already an integer. For example, if \( X+Y \) equals 2.5, the value will be truncated to 2, and the program will branch to line 490, the second line number in the list.

Branching to a line containing a DIM, REM, or DATA statement is not allowed. As many line numbers may be included in an \( \text{ON---GO TO} \) statement as will fit on one line.

**IF—THEN**

At times we want to jump the normal sequence of statements if a certain relationship holds. For this we use an IF--THEN statement, sometimes called a conditional \( \text{GO TO} \) statement. Such a statement occurred at line 130 of MAXSIN. Each IF--THEN statement is of the form

\[
\text{IF formula relation formula THEN line number}
\]

Examples:

\[
\begin{align*}
340 & \text{ IF } \sin(X) \leq M \text{ THEN } 630 \\
120 & \text{ IF } G = 0 \text{ THEN } 165
\end{align*}
\]

The first statement asks whether the sine of \( X \) is less than or equal to \( M \), and tells the computer to go to line 630 if it is. The second statement asks if \( G \) is equal to zero, and tells the computer to go to line 165 if it is. In each case, if the answer to the question is No, the computer will go on to the next line of the program.

**FOR and NEXT**

We have already encountered the \( \text{FOR} \) and \( \text{NEXT} \) statements in loops, and have seen that they go together, one at the entrance to the loop and one at the exit, directing the computer back to the entrance again. Each \( \text{FOR} \) statement is of the form

\[
\text{FOR variable = formula \ TO \ formula \ STEP \ formula}
\]

Most commonly, the formulas will be integers and the \( \text{STEP} \) will be omitted, which means that a step size of plus one is assumed. The accompanying \( \text{NEXT} \) statement is simple in form, but the variable must be exactly the same one as that following \( \text{FOR} \) in the \( \text{FOR} \) statement. The form of the \( \text{NEXT} \) statement is

\[
\text{NEXT variable}
\]

Examples:

\[
\begin{align*}
130 & \text{ FOR } X = 0 \ \text{ TO } 3 \ \text{ STEP } D \\
180 & \text{ NEXT } X \\
120 & \text{ FOR } X4 = (17 + \cos(Z)) / 3 \ \text{ TO } 3 \ast \text{SQR}(10) \ \text{ STEP } 1/4 \\
235 & \text{ NEXT } X4 \\
240 & \text{ FOR } X = 8 \ \text{ TO } 3 \ \text{ STEP } -1 \\
270 & \text{ NEXT } X \\
456 & \text{ FOR } J = -3 \ \text{ TO } 12 \ \text{ STEP } 2 \\
470 & \text{ NEXT } J
\end{align*}
\]

Notice that the step size may be a formula (1/4), a negative number (-1), or a positive number (2). In the example with lines 120 and 235, the successive values of \( X4 \) will be
.25 apart, in increasing order. In the next example, the successive values of X will be 8, 7, 6, 5, 4, 3. In the last example, on successive trips through the loop J will take on values -3, -1, 1, 3, 5, 7, 9, and 11.

If the initial, final, or step size values are given as formulas, the formulas are evaluated once and for all upon entering the FOR statement. The control variable can be changed in the body of the loop. Of course the exit test always uses the latest value of this variable.

If you write 150 FOR Z=2 TO -2 without a negative step size, the loop will not be executed, and the computer will go immediately to the statement following the corresponding NEXT statement.

DIM

Whenever we want to enter a list or table with a subscript greater than 10, we must use a DIM statement to tell the computer to save enough room for the list or table.

Examples:

120 DIM H(35)
135 DIM Q(5,25)

The first statement would enable us to enter a list of 35 items—36 if we use H(0)—and the second a table 5 x 25—or 6 x 26 if we use row 0 and column 0.

END

Every program must have an END statement, and it must be the statement with the highest line number in the program. Its form is simple: a line number with END.

Example:

99 END
2. Advanced BASIC

In Chapter 1, you learned how to write programs in BASIC. In this chapter we will discuss some capabilities of BASIC that were not discussed yet. These include:

- Alphanumeric data and string manipulation
- Files
- Matrices

We will also consider some advanced capabilities in printing output, several functions that we have not yet mentioned, and several statements either in more detail or for the first time. And, finally, we will consider two sample BASIC programs that make use of many of the advanced capabilities of BASIC.

ALPHANUMERIC DATA AND STRING MANIPULATION

Alphanumeric data, names, and other identifying information can be handled in BASIC using string variables. You can enter, store, compare, and print out alphanumeric and certain special characters in the Mark I character set.

A string is any sequence of alphanumeric and certain special characters in the Mark I character set not used for control purposes in the Mark I system. String size is limited to 15 valid characters.

A string variable is denoted by a letter followed by a dollar sign. For example, A$, B$, and X$ denote string variables.

DIM

Strings can be set up as one-dimensional arrays only. If you request a two-dimensional array you will receive the error message DIMENSION TOO LARGE.

Examples:

100 DIM A(5),C$(20),A$(12),D(10,5)
200 DIM R$(35)
300 DIM M$(15),B$(15)

In line 100, only C$ and A$ are string variables. R$, as dimensioned in line 200, will save storage space for 35 fifteen character arrays. Any or all of the 35 strings may in fact be less than fifteen characters long.

LET

Strings and string variables may appear in only two forms of the LET statement. The first is used to replace a string variable with the contents of another string variable.
Example:

156 LET G$=H$

The second is used to assign a string to a string variable.

Example:

160 LET J$="THIS STRING"

Arithmetic operations may not be done on string variables. Requests for addition, subtraction, multiplication, or division involving string variables produce the error message ILLEGAL STRING OPERATION AT XXX.

The LET statement permits multiple variable replacement.

Example:

262 LET X=Y=Z=21*N/2
435 LET A$=G$=J$="THIS STRING"

The first statement places the value of the expression 21*N/2 in variables X, Y, and Z. The second statement assigns the string THIS STRING to variables A$, G$, and J$. Any valid expression or string may be used.

READ and DATA

READ statements can contain string variables intermixed with ordinary variables. In the corresponding DATA statements, every item corresponding to a string variable in the READ statement must be a valid string. If the string contains any characters that have special meaning in BASIC—such as commas, semicolons, leading or trailing spaces, and so on—it must be enclosed in quotation marks. Unquoted strings must begin with an alphabetic character.

Example:

100 READ A$,B$,C$,D$,A,E$
200 DATA "THE","","PEOPLE","","YES--",500,0F THEM.
300 PRINT A$;B$;C$;D$;A;E$
999 END

This program will print out THE PEOPLE, YES-- 500 0F THEM. The DATA statement has quotation marks around B$ because it is a blank space, and around C$ because it includes a comma.

INPUT

Like READ and DATA statements, INPUT statements can contain string variables intermixed with ordinary variables. Every item corresponding to a string variable in the INPUT statement must be a valid string variable. If the string contains characters that have special meaning in BASIC, it must be enclosed by quotation marks. If the string begins with other than an alphabetic character it must be enclosed in quotation marks.

Example:

110 INPUT L$(17),M$,N$(I)
PRINT

The PRINT statement also can contain string variables intermixed with ordinary variables. When a string variable is encountered that has not been assigned, the PRINT statement will produce for that variable a string of 15 blank spaces. A semicolon after a string variable in a PRINT statement causes the printout of the variable following that string to be directly connected to the string variable.

Examples:

135  PRINT A,16,B$,C$;N
140  PRINT 100+L, "DATA",L$;M$;N$
150  PRINT S$

IF—THEN

Only one string variable is allowed on each side of the IF—THEN relation sign. All of the six standard relations are valid (=, <, >, <, >, =, and =>). When strings of different lengths are compared, the shorter string and the corresponding part of the longer string will be used. If they compare, the shorter string is taken to be the lesser of the two.

Examples:

100  IF N$="SMITH" THEN 105
200  IF A$<>B$ THEN 205
300  IF "JUNE"< M$ THEN 305
400  IF D$>= "FRIDAY" THEN 600

You must use quotation marks around the string to be compared, as above, unless it is referenced in the IF—THEN statement by a string variable name.

Characters are compared in their BASIC code representations. The collating sequence used in comparing is listed in Appendix C.

MORE ABOUT PRINTING

Although the format of the printout is automatically supplied for the beginner, the PRINT statement, the TAB function, and image formatted output permit a greater flexibility for the advanced programmer in setting up different formats for his output.

PRINT

The teletypewriter line is divided into five zones of fifteen spaces each. Some control of the use of these zones comes from the use of the comma. A comma is a signal to move to the next print zone or, if the fifth print zone has been used, to move to the first print zone on the next line.

Shorter zones can be made by use of the semicolon. The zones are three spaces long for 1-digit numbers, six spaces long for 2-digit, 3-digit, and 4-digit numbers, nine spaces long for 5-digit, 6-digit, and 7-digit numbers, twelve spaces long for 8-digit, 9-digit, and 10-digit numbers, and fifteen spaces long for 11-digit numbers. As with the comma, a semicolon is a signal to move to the next short print zone or, if the last such zone on the line has been used, to move to the first print zone of the next line.

The first space in any print zone is reserved for the sign, even though it is not printed out if it is plus.
If you typed the program

```
100 FOR I=1 TO 15
110 PRINT I
120 NEXT I
999 END
```

the teletypewriter would print 1 at the beginning of the first line, 2 at the beginning of the next line, and so on, finally printing 15 at the beginning of the fifteenth line. But if you changed line 110 to read

```
110 PRINT I,
```

you would have the numbers printed in zones, reading

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

If you wanted the numbers printed in the same fashion, but more tightly packed, you could change line 110 to replace the comma with a semicolon

```
110 PRINT I;
```

and the result would be printed

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

You should remember that a label inside quotation marks is printed just as it appears, and also that the end of a PRINT statement signals a new line, unless a comma or semicolon is the last symbol. The instruction

```
150 PRINT X,Y
```

will result in the printing of two numbers and the return to the next line, but

```
150 PRINT X, Y,
```

will result in the printing of two numbers and no return. The next number to be printed will be printed in the third zone on the same line as the values of X and Y.

Since the end of a PRINT statement signals a new line, a statement such as

```
250 PRINT
```

will cause the teletypewriter to advance the paper one line. It will put a blank line in your printout, if you want to use it for vertical spacing of your results, or it will cause the completion of a partly filled line, as illustrated in the following part of a program:

```
100 FOR M=1 TO N
110 FOR J=0 TO M
120 PRINT B(M,J);
130 NEXT J
140 PRINT
150 NEXT M
```

The program will print \( B(1,0) \) and next to it \( B(1,1) \). Without line 140, the teletypewriter would then go on printing \( B(2,0) \), \( B(2,1) \), and \( B(2,2) \) on the same line, and even \( B(3,0) \), \( B(3,1) \), and so on, if there were room. Line 140 directs the teletypewriter to start a new line after printing the \( B(1,1) \) value corresponding to \( M=1 \), and to do the same thing after printing the value of \( B(2,2) \) corresponding to \( M=2 \), and so on.
TAB

The print function TAB permits tabbing of the teletypewriter. Whenever the TAB function is used in the PRINT statement, it will cause the print head to move over to the position indicated by the argument of TAB.

Example:

150 PRINT X; TAB(10); Y; TAB(2*N); Z

The argument of TAB refers to a print position on the teletypewriter line. The positions are assumed to run from 0 through 74. In the example, if the value of N is 10, the print head will move to the 10th print position after printing the value of X, and to the 20th print position after printing the value of Y.

When using the TAB function, you should use the semicolon in the PRINT statement in order to minimize field width.

If the argument of TAB is less than the current teletypewriter print head position, it is ignored.

All arguments of TAB are treated modulo 75.

Rules for Printing Numbers

The following rules for the printing of numbers will help you in interpreting printed results.

- If a number is an integer, the decimal point is not printed. If the integer is larger than or equal to \(2^{30}\) (i.e. 1,073,741,824), the teletypewriter will print the first digit, followed by (1) a decimal point, (2) the next five digits, and (3) an E followed by the appropriate exponent integer. For example, it will print 32,437,580,259 as 3.24376E+10.

- For any decimal number, no more than six significant digits are printed.

- For a number less than 0.1, the E notation is used unless the entire significant part of the number can be printed as a six decimal number. For example, .03456 means that the number is exactly .0345600000, but 3.45600E-2 means that the number has been rounded to .0345600.

- Trailing zeroes after the decimal point are not printed.

The following program, in which we print out the first 45 powers of 2, shows how numbers are printed.

```
100 FOR I=1 TO 45
110 PRINT 2^I;
120 NEXT I
999 END
RUN
```

```
PRINT 14:53
2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144 524288 1048576 2097152 4194304 8388608 16777216 33554432 67108864 134217728 268435456 536870912 1073741824 2147483648 4294967296 8589934592 17179869184 34359738368 68719476736 137438953472 274877906944 549755813888 1099511627776 2199023255552 4398046511104 8796093022208 17592186044416 35184372088832 70368744177664 140737488355328 281474976710656 562949953421312 1125899906842624 2251799813685248 4503599627370496 9007199254740992 18014398509381984 36028797018763968 72057594037527936 144115188075055872 288230376150111744 576460752300223488 1152921504600446976 2305843009201893952 4611686018403787904 9223372036807575808
```

23
You can set up formatted line output by use of the PRINT USING and image statements.

The form of the PRINT USING statement is

```
PRINT USING line number, output list
```

where the line number is that of the image statement to be used in formatting the output line, and the output list can consist of numbers, variables, string constants, string variables, and functions.

The form of the image statement is

```
line number:line image
```

where the line number is that of the image statement in the program, and the line image consists of format control characters and printable constants.

Format control characters are

- `'` (apostrophe) a one-character field that is filled with the first character in an alphanumeric string regardless of the string length.
- `" "` (quotation marks) the replacement field of an alphanumeric string of two or more characters; the field width includes the quotation marks as well as the characters (if any) contained within the marks.
- `#` (pound sign) the replacement field for a numeric character.
- `####` (four up-arrows) indicates scientific notation for a numeric field.

All other characters are treated as printable constants.

The following simple example is part of a program, showing the use of the PRINT USING statement, the line image statement, and format control characters.

Example:

```
110 PRINT USING 120,AS,"$",A,324,X
120: " " '####' ####### 
```

If the values of A$, A, and X are FIRST, 12.9, and 24687, output is

```
FIRST $ 12.90 324 2.47E+04
```

An image statement must begin with a colon. It is composed of fields which form the print line. There are five types of fields:

- Integer fields
- Decimal fields
- Exponential fields
- Alphanumeric fields
- Literal fields
Integer Fields

The following rules apply to integer fields.

- An integer field is composed of pound signs (#).
- Numbers in an integer field are right justified and truncated if they are not integral.
- The field will be widened to the right if the number is too big.
- The field must reserve a place for the algebraic sign.
- If the number is greater than $1,073,741,823$ in absolute value, an asterisk will be printed.

Example:

```
100: #### #### ####
110 READ A,B,C
120 PRINT USING 100,A,B,C
130 G0 T0 110
140 DATA 123,45,,-34.856,457,34,,-17,89,999
999 END
RUN

PRINTU 15:00
   123   457   -34   -17   89

OUT OF DATA IN 110
```

Decimal Fields

The following rules apply to decimal fields.

- A decimal field is a string of pound signs (#) with an imbedded period. Note that .### is not a decimal field because the period is not imbedded.
- The number will be rounded to the number of places specified by the pound signs following the decimal.
- The number is right justified, placing the decimal as given in the field definition.
- The field will be widened to the right if the number is too large.
- The field must include a place for the algebraic sign.

Example:

```
100: #### #### #### #### #### #### #### ####
110 READ A,B,C,D
120 PRINT USING 100,A,B,C,D
130 G0 T0 110
140 DATA 123.456,-34.856,47.5,-0177
150 DATA 1.999,876.55,-17,.893
999 END
RUN

PRINTU 15:03
   123.46   457.89   -34.8560   48.0018
   2.0000   676.5500   -17.893

OUT OF DATA IN 110
```
Exponential Fields

The following rules apply to exponential fields.

• An exponential field is a decimal field followed by four up-arrows(↑), which reserve
  a place for the exponent.

• The pound signs preceding the decimal represent the factor by which the exponent
  will be adjusted.

• The number will be rounded as with decimal fields.

• A place must be reserved for the sign.

Example:

```
1001 #.########  #.########  #.#....  #.#....
110 READ A$B$C$D
120 PRINT USING 100,A$B$C$D
130 G0 T0 110
140 DATA 123.456,-34.856e47,7,-0177
150 DATA 1.999e876.5e-55,-17,-893
999 END
RUN
PRINTU 15:05
  .12346E+03  -3.486E+01  48.E+00  -.18E-01
  .19990E+01  8.766E+02  -17.E+00  +89E+00
OUT OF DATA IN 110
```

Alphanumeric Fields

The following rules apply to alphanumeric fields.

• The apostrophe is used to print the first character from a string variable or quoted
  constant.

• A field bounded by quotation marks is used to print two or more characters.

• In an alphanumeric field of two or more characters, the string is left justified within
  the field and blank filled or truncated on the right.

Example:

```
100: "23456" "THE NAME GOES HERE"  '  '  '
110 READ A$B$C$D$E$
120 PRINT USING 100,A$B$C$D$E$
130 DATA ABCDEFGHI
140 DATA ABCDEFG
150 DATA ABC
160 DATA ABC
170 DATA ABC
999 END
RUN
PRINTU 15:08
  ABCDEFG  ABCDEF  A  AA
```
Literal Fields

A literal field is composed of characters or character strings that are not control characters. It will appear on the print line exactly as it appears in the image.

Example:

```
100: THE VALUE FOR A IS "####.##"
110 LET A=100.54
120 LET AS="$"
130 PRINT USING 100, AS, A
999 END
RUN

PRINTU 15:09
THE VALUE FOR A IS $ 100.54
```

General Rules

The following rules apply in general to formatted line output.

- A program may contain up to 100 images.

- The list elements in the PRINT USING statement may be expressions, variables, numeric constants, and quoted literals.

- Numeric list elements must replace numeric fields, and alphanumeric elements must replace alphanumeric fields, or you will receive the error message BAD IMAGE.

- If the output list contains more elements than there are replaceable fields in the image statement, a carriage return is supplied after the last field in the image, and the image is reused. The extra elements will be printed on a second line only if they match the image fields that are to be used.

Example:

```
100 PRINT USING 120
110 PRINT
120: I I+2 I+3
130: """"""""""""""""""""""""""""""""
140 FOR I=1 TO 6
150 LET A(I)=I
160 LET B(I)=I+2
170 LET C(I)=I+3
175 NEXT I
180 FOR I=1 TO 6 STEP 2
190 PRINT USING 130, A(I), B(I), C(I), A(I+1), B(I+1), C(I+1)
200 NEXT I
999 END
RUN

PRINTU 15:14
I I+2 I+3
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
```
The following program demonstrates one kind of application in which the formatted output line is useful.

Example:

```
100 PRINT USING 170
110 PRINT
120 FOR I=1 TO 4
130 READ A$ , A , B
140 LET T=A*B
150 PRINT USING 180 , A$ , A , B , "$" , T
160 NEXT I
170: NAME
180:"
190 DATA ANDREWS, 47.50, 3.987, KELLY, 40, 2.865, MANLEY, 46, 3.020
200 DATA ZUMPANO, 42.34, 4.255
999 END
RUN
WAIT.
```

**FORMAT**

<table>
<thead>
<tr>
<th>NAME</th>
<th>HRS WORKED</th>
<th>RATE</th>
<th>PAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDREWS</td>
<td>47.50</td>
<td>3.987/HR</td>
<td>$ 189.38</td>
</tr>
<tr>
<td>KELLY</td>
<td>40.00</td>
<td>2.865/HR</td>
<td>$ 114.60</td>
</tr>
<tr>
<td>MANLEY</td>
<td>46.00</td>
<td>3.020/HR</td>
<td>$ 138.92</td>
</tr>
<tr>
<td>ZUMPANO</td>
<td>42.34</td>
<td>4.255/HR</td>
<td>$ 180.16</td>
</tr>
</tbody>
</table>

**FUNCTIONS**

There are two functions that were listed in Chapter 1 but not described: INT and RND.

Three other functions that you will sometimes find useful are SGN, CLK, and TIM. And you can write your own functions by use of the DEF statement.

**INT**

The INT function is the one that frequently appears in algebraic computation as \([x]\), and it gives the greatest integer not greater than \(x\). Thus INT(2.35) equals 2, INT(-2.35) equals -3, and INT(12) equals 12.

One use of the INT function is to round numbers. We can use it to round to the nearest integer by asking for INT(X+.5). This will round 2.9, for example, to 3, by finding

\[
\text{INT}(2.9+.5) = \text{INT}(3.4) = 3
\]

You should convince yourself that INT(X+.5) will do the rounding guaranteed for it, that it will round a number midway between two integers up to the larger of the integers.

It can also be used to round to any specific number of decimal places. For example, \(\text{INT}(10*X+.5)/10\) will round \(X\) correct to one decimal place; \(\text{INT}(100*X+.5)/100\) will round \(X\) correct to two decimal places; and \(\text{INT}(X*10^D+.5)/10^D\) will round \(X\) correct to \(D\) decimal places.

**RND**

The function RND is a pseudo random number generator. It requires a single argument, which has the following meanings:
• If the argument is positive, the argument is used to initiate the random number sequence.

• If the argument is negative, a random number is used to initiate the random number sequence.

• If the argument is zero, RND will supply a random number. The first use of RND(0) in a program will always yield the same random number.

A positive or negative argument would probably be used to initiate a sequence of random numbers, after which a zero argument would be used repeatedly.

If the initial value used for the argument is any power of 2, the same initial random number results as when 2 is used.

If we want the first twenty random numbers, we can use the following program to get twenty six-digit decimals.

Example:

```
100 LET X=RND(1)
110 FOR L=1 TO 20
120 PRINT RND(0),
130 NEXT L
999 END
RUN
```

```
RNDTST  15123
.473599  .442519  .498805  .373168  .921321
.123978  .68505E-02  .406526E-02  .341097  .468896
2.97623E-02  .75441  .498551  .242898  9.31652E-02
.280064  .159309  .211611  .042684  .383241
```

If, on the other hand, we want twenty random one-digit integers, we can change line 120 to read

```
120 PRINT INT(10*RND(0))
```

and we then obtain

```
RNDTST  15124
4 4 4 3 9 1 0 0 3 4 0 7 4 2 0 2 1 2 0 3
```

We can vary the kind of random numbers we get. For example, if we want twenty random numbers ranging from 1 to 9 inclusive, we can change line 120 as shown below

```
120 PRINT INT(9*RND(0)+1))
```

```
RNDTST  15125
5 4 5 4 9 2 1 1 4 5 1 7 5 3 1 3 2 2 1 4
```

Or we can obtain random numbers which are integers from 5 to 24 inclusive by changing line 120 as follows
In general, if we want our random numbers to be chosen from \( A \) integers of which \( B \) is the smallest, we would call for
\[
\text{INT}(A \cdot \text{RND}(0) + B)
\]
after first having initiated the random number sequence with a positive or negative argument, as in line 100 of our sample program.

If you were to run the first version of our sample program again, you would get the same twenty numbers in the same order. But we can get a different set by throwing away some of the random numbers. In the following program we find the first ten random numbers and do nothing with them. We then find the next twenty and print them. You can see, by comparing this with the earlier program, that the first ten of these random numbers are the same as the second ten of the first program.

Example:

```plaintext
100 LET Z=RND(1)
110 FOR I=1 TO 10
120 LET Y=RND(0)
130 NEXT I
140 FOR I=1 TO 20
150 PRINT RND(0),
160 NEXT I
999 END
```

Run

```
RNDTST    15:26
14  13  14  12  23  7  6  5  11  14  5  20  14  9  6
  10  8  9  5  12
```

The function `SGN` allows you to test for the sign of any value. The form is `SGN(argument)` and it yields +1, -1, or 0 depending on the value of the argument. The options are

<table>
<thead>
<tr>
<th>Argument Value</th>
<th>Yields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0</td>
</tr>
<tr>
<td>Positive, not zero</td>
<td>+1</td>
</tr>
<tr>
<td>Negative, not zero</td>
<td>-1</td>
</tr>
</tbody>
</table>

**Examples**

- `SGN(0)` yields 0
- `SGN(-1.82)` yields -1
- `SGN(989)` yields +1
- `SGN(-0.001)` yields -1
- `SGN(-0)` yields 0
CLK

The function CLK(X), X being a dummy argument, yields the time of day in military hours.

Examples:

100 PRINT CLK(X)
295 IF CLK(X) > 15.00 THEN 1000
130 IF A-CLK(X) < .5 THEN 1000

TIM

The function TIM(X), X being a dummy argument, yields the program elapsed time in seconds.

Examples:

100 PRINT TIM(X)
688 IF TIM(X) > 10 THEN 1000

DEF

In addition to making use of the standard functions, you can define any other function that you expect to use several times in your program. You use a DEF statement to define such a function. The name of the defined function must be three letters, the first two of which are FN. Hence you can define up to 26 functions in one program: FNA, FNB, FNC, and so on.

The usefulness of DEF you can see in a program, for example, where you often need the function $e^{-x^2}$. You introduce the function by the line

130 DEF FNE(X) = EXP(-X^2)

and later on call for various values of the function by FNE(.1), FNE(3.45), FNE(A+2), and so on. DEF can be a great time-saver when you want values of some function for a number of different values of the variable.

The DEF statement may be put anywhere in the program, and the expression to the right of the equal sign may be any formula that can be fitted on one line. It may include any combination of other functions, including ones defined by other DEF statements, and it can involve other variables besides the one denoting the argument of the function. For example, if FNR is defined by

170 DEF FNR(X) = SQR(2+LOG(X)-EXP(Y*Z)*(X+SIN(2*Z)))

and you have previously assigned values to Y and Z, you can ask for FNR(2.175). You can give new values to Y and Z before the next use of FNR, if you want to.

DEF is generally limited to cases where the value of the function can be computed within a single BASIC statement. Often much more complicated functions, or even pieces of a program, must be calculated at several different points within a program. For these functions, the GOSUB statement will frequently be useful. It is described in the next section.

ADDITIONAL BASIC STATEMENTS

Several kinds of BASIC statements were not covered in Chapter 1. These are discussed in this section. They are:
GO SUB and RETURN

When a particular part of a program is to be used more than one time, or possibly at several different places in the overall program, it is most efficiently programmed as a subroutine. The subroutine is entered with a GO SUB statement.

Example:

190 GO SUB 310

The line number, 310, is the line number of the first statement in the subroutine.

The last line of the subroutine should be a RETURN statement, directing the computer to return to the earlier part of the program.

Example:

450 RETURN

This statement, if it is the last line in the subroutine entered in the previous example, tells the computer to go back to the first line numbered greater than 190 and continue the program there.

You may use a GO SUB statement inside a subroutine to execute yet another subroutine. This is called nested GO SUBs. It is absolutely necessary that a subroutine be left only with a RETURN statement. Using a GO TO or an IF--THEN to get out of a subroutine will not work correctly. You may have several RETURNS in the subroutine so long as only one of them will be used.

You should be very careful not to write a program in which a GO SUB appears inside a subroutine that refers to one of the subroutines already entered. Recursion is not allowed.

The following example, a program for determining the greatest common divisor of three integers using the Euclidean Algorithm, illustrates the use of a subroutine.

Example:

100 PRINT "A","B","C","GCD"
110 READ A,B,C
120 LET X=A
130 LET Y=B
140 GO SUB 230
150 LET X=G
160 LET Y=C
170 GO SUB 230
180 PRINT A,B,C,G
190 GOTO 110
200 DATA 60,90,120
210 DATA 38456,64872,98765
220 DATA 32,384,72
230 LET G=INT(X/Y)
240 LET R=X-G*Y
250 IF R=0 THEN 290
260 LET X=Y
270 LET Y=R
The first two numbers are selected in lines 120 and 130, and their GCD is determined in the subroutine, lines 230 - 300. The GCD just found is called X in line 150, the third number is called Y in line 160, and the subroutine is entered from line 170 to find the GCD of these two numbers. This number, the GCD of the three given numbers, is printed out with the three numbers in line 180.

CALL

The CALL statement is used to call an external program for use as a subroutine within the main program just as the GOSUB statement calls a subroutine inside the main program. The statement form is CALL saved program name.

Examples:

100 CALL HISDWN
200 CALL EQCLS*

You can call either previously saved programs of your own, as in line 100; common programs in your catalog library, as in line 200; or system library programs, either regular or run-only.

The standard program naming rules apply.

Examples:

140 CALL A B
150 CALL AB

Statements 140 and 150 both call a program named AB, since BASIC ignores all leading, trailing, and imbedded blanks. No arguments are permitted after the program name in the CALL statement. Subroutines may call other routines, but no program may call the main program or itself.

The return from a subroutine to the calling program is by a RETURN statement. Multiple returns are permissible. The return is always to the statement immediately following the statement in which the program was called.

All variables and defined functions are common to the main program and the called subroutines. They need not be defined separately in each program.
Statement 110 calls DEFPRT, which stores 4 in Y and 7 in X, and calculates a value for A by use of a function defined in line 100 of MAIN. The function uses the value 3 for Z as defined in statement 220 of DEFPRT. DEFPRT then returns to the statement immediately following the CALL statement, and the calculated value for A, 8.60233, is printed.

An END or STOP statement terminates all execution, whether it is executed in a subroutine or in the main program.

The line numbers in the different programs are completely independent. GOTO and IF–THEN statements reference line numbers in their own program only.

Data is compiled from the main program first, and then from each of the called programs in the order in which the CALL statements are encountered.

Consider the following example.

Example:

```
NEW
NEW FILE NAME--DEFPRT
READY*

100 LET Y=4
110 LET X=7
220 LET A=FNP(3)
230 RETURN
999 END
SAVE
WAIT*

READY*

NEW
NEW FILE NAME--MAIN
READY*

100 DEF FNPCZ)=SQRT(X^2+Y^2+Z^2)
110 CALL DEFPRT
120 PRINT A
999 END
RUN
WAIT*

MAIN 15:35

8.60233
```
Statement 100 reads the numbers 1 and 2 into A and B. Statement 110 transfers control to SUBR, which reads 3, 4, and 5 (not 6, 7, and 8) into variables X, Y, and Z. After the return to statement 120, 6, 7, and 8 are read into C, D, and E.

The CALL statement allows more effective use of program space available. A program referred to by a CALL statement will be compiled only once no matter how many times it is called in each of the routines. Object code generated by the called program counts toward object code limitation, but the characters in the called program do not count toward the BASIC character limitation.

As many as 10 different programs may be called.

**STOP**

The **STOP** statement is equivalent to **G0 T0 XXXX**, where XXXXX is the line number of the **END** statement in the program. It is useful in programs having more than one natural finishing point. For example, the following two parts of programs are exactly equivalent.

Example:

```
250 G0 T0 999
   ...
340 G0 T0 999
   ...
999 END
```

```
250 STOP
   ...
340 STOP
   ...
999 END
```

**REM**

The **REM** statement allows you to insert explanatory remarks in a program. The form is **REM any comment**. The computer completely ignores the part of the line following **REM**, allowing you to include directions for using the program, identifications of the parts of a long program, or anything else. Although what follows **REM** is ignored, you may use the line number of a **REM** statement in a **G0SUB** or **IF--THEN** statement.
Examples:

```
100 REM INSERT DATA IN LINES 900-998. THE FIRST
110 REM NUMBER IS N, THE NUMBER OF POINTS. THEN
120 REM THE DATA POINTS THEMSELVES ARE ENTERED, BY

200 REM THIS IS A SUBROUTINE FOR SOLVING EQUATIONS.
:::
300 RETURN
:::
520 GOSUB 200
```

RESTORE

Sometimes it is necessary to use data in a program more than once. The RESTORE statement permits reading the data as many additional times as necessary. Whenever RESTORE is encountered in a program, the computer restores the data block pointer to the first item of data. A subsequent READ statement will then start reading the data all over again.

One word of warning: if the data items you wish to use again are preceded by code numbers or parameters, superfluous READ statements should be used to pass over the numbers.

As an example, the following part of a program reads the data, restores the data block to its original state, and reads the data again. Note the use of line 570 to pass over the value of N, which is already known.

Example:

```
100 READ N
110 FOR I=1 TO N
120 READ X
:::
200 NEXT I
:::
560 RESTORE
570 READ X
580 FOR I=1 TO N
590 READ X
```

CHAIN

The CHAIN statement allows you to stop the execution of the current program and begin compilation and execution of another program without direct intervention. The CHAIN statement is equivalent to giving the commands STOP, OLD, a program name, and RUN.

The statement form is

```
CHAIN saved program name
```

or

```
CHAIN saved program name, line number
```

Only one program name may appear in a statement. The name must conform to the rules used in naming BASIC programs.
Examples:

100 CHAIN NEXT
100 CHAIN NEXT, 100
100 CHAIN PLÔTER***
100 CHAIN TUT01S***
100 CHAIN PAYRÔL, 555

Notice that, as shown in the examples, BASIC library programs may be chained.

When a number appears after the saved program name, as in the second and fifth lines of the examples, the number indicates the line number of the named program at which execution is to begin. When no number appears, execution begins with the first executable statement of the named program.

Once a CHAIN statement is executed, the current program is stopped and the named program brought in. Because there is no logic path to any statements following the CHAIN statement, all needed current program statements must be executed before the CHAIN statement.

DATA FILES

An external data file is a saved program in which you can record information for later use. There are two types of data files: BCD and binary. The data in a file must all be of one type.

BCD data files may be created with a BASIC program or they may be typed in as a saved program. Thus all of the editing commands, such as LIST, EDIT DELETE, and so on, may be used in accessing and modifying BCD files.

Binary data files can be created only with a BASIC program. They cannot be created from a terminal or altered in the same manner as BCD files.

If a BCD or binary file is to be written during program execution, an area on the disc large enough to contain the entire file to be generated must be reserved in your library before program execution. This is done by renaming and saving special files available in the system library (see "Dummy Catalog Files"). These files are empty. Their purpose is to preset the size of the files to be generated.

Example:

OLD
OLD FILE NAME--CH0768***
READY
RENAME
NEW FILE NAME--WFILE
READY
SAVE
READY

In the example, the data file WFILE has been created; it has a storage capacity of 768 characters.

File Reference

The form of the file reference statement is

```
FILES name 1; name 2; ...; name n
```

where name 1, name 2, and so on are the names of files to be read or written by the program. The file reference statements must precede all executable statements in the program. The
number of files that can be used in a program depends on the size of the program, but you can always reference at least 8 files in any program.

The named files must be saved in your catalog before running the program. Files referenced but not saved produce the error message FILE NOT SAVED when you run the program.

File naming must conform to the established conventions for naming programs, except:

- File names must not contain semicolons. They will be interpreted as file name separators.
- Leading and imbedded blanks are ignored.
- The file name is left-justified.
- File names should not contain slashes or commas.

Examples:

100 FILES A;B;C  
   or

100 FILES A
110 FILES B;C

The following file names are identical:

100 FILES XYZ
100 FILES XY Z
100 FILES X Y Z

File Designator

The file designator is used in all statements referencing files. It singles out a file named in the file reference statement. It may be an integer, an expression, a variable, or a subscripted variable.

Example:

100 FILES P;Q

170 READ #1,A(I), B(I)

195 READ #A*B, A(I), B(I)

Statement 170 refers to file P. Statement 195 refers to a file depending on the value of the expression A*B, which must be an integer. If the value of A*B is 1, file P is selected; if the value of A*B is 2, file Q is selected.

If the value of the file designator is less than one, non-integral, or greater than the number of files referenced, the error message ILLEGAL FILE DESIGNATOR will be printed.

Dummy Catalog Files

Since it is necessary to save a file of a size large enough to contain the data to be written, the system library contains the following six files which you can rename and save in your catalog.
The number of data points that can be written into a file is a function of the number of characters in each data point. Line numbers and spaces must also be considered in the count.

### BCD DATA FILES

BCD files are sequential access files. For each execution of the READ or WRITE statement, data is transmitted serially. BCD files contain line numbers and are listable.

If a BCD file with initial values is to be read from the disc, you must prepare it before program execution and save it in your catalog.

**Example:**

```
NEW
NEW FILE NAME--RFILE
READY
100 1,1.5,2,2.5,3
110 3.5,4,4.5,5,5.5
SAVE
READY
```

When preparing a BCD file, you may if you wish use a blank in place of a comma as a data separator. This option allows files prepared in FORTRAN to be processed in BASIC and conversely. RFILE in the previous example can be prepared as:

```
100 1 1.5 2 2.5 3
110 3.5 4 4.5 5 5.5
```

Note that the word DATA is not needed in these files. The first number on each line is the line number.

All files are in either read mode or write mode. Initially, the FILES statement results in all files being set to the read mode. This protects you from accidentally destroying valuable files. You can later change the mode of any file by issuing a SCRATCH or RESTORE statement. The SCRATCH statement establishes the designated file as write mode. The RESTORE statement establishes the designated file as read mode.

### File Reading

The form of the BCD read file statement is

```
READ #file designator, input list
```

where the file designator is as previously described. The pound sign denotes a BCD file.

The input list consists of the variables, separated by commas, into which the data is to be read. The list may contain non-string and string variables, and any of them may be subscripted.
Example:

A file containing 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5 is to be read into A(I) and B(I).
100 FILES RFILE
110 FOR I=1 TO 5
120 READ #1,A(I),B(I)
130 NEXT I

For each execution of the READ # statement one value is read into an A(I) and B(I) so that, at the termination of the loop:

\[
\begin{align*}
A(1) &= 1 & B(1) &= 1.5 \\
A(2) &= 2 & B(2) &= 2.5 \\
A(3) &= 3 & B(3) &= 3.5 \\
A(4) &= 4 & B(4) &= 4.5 \\
A(5) &= 5 & B(5) &= 5.5
\end{align*}
\]

The file pointer will remain positioned following the last read data item (5.5 in the example) until further file statements designating the file (RFILE in the example) are executed.

File Writing

The form of the BCD write file statement is

\[
\text{WRITE #file designator, output list}
\]

where the file designator is as previously described.

The output list consists of the variables, separated by semicolons, from which the file is generated. The list may contain non-string variables, string variables, strings, and expressions. Subscripting is permissible in the output list.

WRITE # always generates a file beginning with line number 1000, incrementing by 10 for each new line. The line number sequencing can be modified, if you wish, by EDIT RESEQUENCE.

Each WRITE # statement generates one line of output unless the teletypewriter line limit is exceeded or the last list item is followed by a semicolon. When the teletypewriter line limit is exceeded, writing continues on the next line with the next data item. When the output list is followed by a semicolon, subsequent writing occurs on the same line in a closely packed format.

Example:

100 FILES WFILE
110 SCRATCH #1
120 FOR I=1 TO 25
130 WRITE #1,I;I*I
140 NEXT I

When listed WFILE would contain

\[
\begin{align*}
1000 &= 1 & 1 \\
1010 &= 2 & 4 \\
1020 &= 3 & 9 \\
\cdots &= \cdots & \cdots \\
1240 &= 25 & 625
\end{align*}
\]
Following is an example showing how strings and string variables are used with files.

Example:

```
100 FILES STRING
110 SCRATCH #1
120 LET A$="STRING1"
130 WRITE #1,A$;"STRING2"
140 WRITE #1,"STRING2";A$
```

Then listing the file STRING gives

```
1000 STRING1 STRING2
1010 STRING2 STRING1
```

**End-of-File and End-of-Space**

The end-of-file (EOF) is a special mark written by BASIC itself that indicates the end of data in the file.

The end-of-space (EOS) is the physical end of the disc area reserved for a file. When a file is completely filled with data, EOF = EOS; otherwise EOF < EOS.

Whenever a word is generated with a WRITE # statement, an EOF mark is placed immediately following the last word written. Subsequent transmissions to the file move the EOF mark so that it always follows the last word written. When a file is generated from the teletypewriter, an EOF mark is placed immediately after the last data item in the file as soon as the file is saved.

**End-of-File Test**

The form of the statement that tests for an EOF mark or the EOS is

```
IF END #file designator THEN line number
```

where the file designator is as previously described.

The statement will test whether an EOF was detected by the last command reading the designated file. When writing a file, the statement will test whether an EOS was detected.

If the last READ # statement found an EOF mark, the program will go to the line number specified in the IF END # statement. Otherwise the next sequential statement is executed.

If the program continues reading or writing a file after the EOF or EOS has been detected, an error message (END OF FILE or END OF SPACE) will be printed and the program will continue. The error message will be printed out each time an attempt is made to exceed the EOF or EOS limit.

Example:

```
100 FILES F1;F2
110 SCRATCH #2
120 DIM X(100),Y(100),Z(100)
130 FOR I=1 TO 100
140 IF END #1 THEN 180
150 READ #1,X(I),Y(I),Z(I)
160 WRITE #2,SQR(X(I)^2+Y(I)^2+Z(I)^2)
170 NEXT I
180 STOP
```
If file F1 contained 300 data items, no EOF would be encountered; but if it contained only 150 data items, for example, the IF END # statement (line 140) would cause a transfer to line 180 following the 50th execution of the loop.

File Restoring

The form of the BCD restore file statement is

    RESTORE #file designator

where the file designator is as previously described.

The RESTORE # statement causes the position of the designated file pointer to be moved so that the next transmission is from the beginning of the file.

If a file is already restored, the RESTORE # statement merely sets the file to read mode.

BASIC automatically restores the referenced files before a program begins executing.

Example:

100 FILES UTIL
110 SCRATCH #1
120 FOR I=1 TO 50
130 WRITE #1,I;SQR(I)
140 NEXT I
150 RESTORE #1
160
170 READ #1,X,X1

In the example the RESTORE # statement (line 150) is required to restore the written file before reading it. Reading then begins at the first data item in the file.

A file being written cannot be read before it is restored. A file being read cannot be written before it is scratched. This means that if any modifications are to be made to an existing file via the program, it must be copied (written) to another file up to the modification point. The modification can then be written into the second file.

File Scratching

The form of the BCD scratch file statement is

    SCRATCH #file designator

where the file designator is as previously described. This statement causes the designated file to be scratched, or made ready for writing. If the file is already reset or restored the statement merely sets the file to write mode. Following the SCRATCH # statement, transmission to the file commences at the beginning of the file.

Example:

100 FILES UTIL
110 SCRATCH #1
120 FOR I=1 TO 50
130 WRITE #1,RND(0)
140 NEXT I
150 RESTORE #1
160 READ #1,X,Y,Z
In the example, the SCRATCH # statement (line 110) must be executed before the write into the file. The RESTORE # statement must be executed before reading from the written file.

File Backspacing

The form of the BCD backspace file statement is

```
BACKSPACE #file designator
```

where the file designator is as previously described. This statement is permitted only on files in the read mode.

The BACKSPACE # statement causes the position of the data pointer for the file being read to be moved backward one data item. If the data pointer is already at the beginning of the file, the backspace statement is ignored.

Some applications require a file to be processed forward and then backward. The following example illustrates how this can be done.

Example:

```
100 FILES F1
110 IF END #1 THEN 210
120 READ #1,A
130 REM COUNT NUMBER OF POINTS IN FILE
140 LET N=N+1
  .
  .
200 G0 TO 110
210 FOR I=1 TO N
220 BACKSPACE #1
230 READ #1,A
240 BACKSPACE #1
  .
  .
300 NEXT I
```

BINARY DATA FILES

Binary data files are not listable. The file space normally used for line numbers in files can be used for data. They are random access files; that is, data can be written into or read from a binary file in any order. Since data conversion routines are eliminated in processing binary files, program efficiency is increased.

For binary files, the restrictions on file mode are eliminated. Binary files may be read or written without having been previously restored or scratched. The RESTORE and SCRATCH statements serve only to move the data element pointer to the first element in the file.

File Writing

The form of the binary write file statement is

```
WRITE : file designator, output list
```

where the file designator is as previously described. The colon denotes a binary file.
The output list may contain numeric variables, string variables, or literals. The numeric variable or literal will cause a two-word entry into the file. The string variable or literal will cause a six-word entry into the file. Each two-word entry is defined as a data element. Consequently each numeric variable or numeric literal consists of one data element, and each string variable or string literal consists of three data elements.

For example, a numeric variable or literal may appear in a WRITE : statement as:

```
100 WRITE:1,A        (numeric variable)
110 WRITE:1,10       (numeric literal)
```

A string variable or literal may appear in a WRITE : statement as:

```
120 WRITE:1,A$       (string variable)
130 WRITE:1,"ASTRN"  (string literal)
```

The following diagram illustrates how each of these will be entered into the file. In the diagram, each block represents one computer word.

```
A 10 A$ ASTRN

1 2 3 4 5 6 7 8
```

Binary files contain no line numbers. Data may be written into the file sequentially or randomly.

The following is an example of a sequentially created file.

Example:
```
100 FILES WFILE
110 SCRATCH:1
120 FOR I=1 TO 25
130 WRITE:1,I;I*I
140 NEXT I
```

File WFILE will contain 25 pairs of numbers, or 50 successive data elements.

In the following example, string information is written into a binary file.

Example:
```
100 FILES STRING
110 SCRATCH:1
120 LET M$="MSTRING"
130 WRITE:1,M$;"NSTRING"
```

File STRING will contain two strings or six data elements, three data elements for each string.
File Reading

The form of the binary read file statement is

```
READ : file designator, input list
```

where the file designator is as previously described.

The input list may contain numeric variables or string variables. The numeric variable will cause one data element (two words) to be read from the file. The string variable will cause three data elements (six words) to be read from the file.

Data may be read sequentially or randomly from a file.

In the following example data is read sequentially.

Example:

```
100 FILES RFILE
110 FOR I=1 TO 5
120 READ:1,A(I),B(I)
130 NEXT I
```

The first ten data elements in file RFILE will be read alternately into arrays A and B.

Random Accessing

The form of the random access statement is

```
SET : file designator, variable
```

where the file designator is as previously described.

The SET : statement will position the pointer of the designated file to the element specified by the variable. The variable may be an integer or an expression. If the value of the variable is less than one, non-integral, or greater than the length of the file, the error message ILLEGAL POINTER will be printed out.

Example:

```
100 FILES VFIL
110 SCRATCH:1
120 FOR I=1 TO 25
130 WRITE:1,I*I
140 NEXT I
150 SET:1,7
160 READ:1,X,Y
```

File VFIL will contain 25 elements. In line 150, the pointer will be positioned to the 7th element. Reading then begins at the 7th element, and the values 49 and 64 will be read into X and Y.

If the file being processed contains string variables, the length of a string entry must be considered before the SET : statement is used.

Example:

```
100 FILES RNF
110 SCRATCH:1
120 WRITE:1,“ASTRN”;“BSTRN”
130 SET:1,4
140 READ:1,B$
```
When the WRITE : statement is executed, ASTRN will occupy the first three entries of file RNF, and BSTRN will occupy the next three entries. To access the second string, the pointer must be set to the position where the second string begins, which is done by line 130. The string data, BSTRN, is then read into the string variable, B$.

The following uses of the SET : statement are acceptable.

```
180 SET:1,A*B
195 SET:3,A(I)
```

In each case the value of the variable must be an integer within the limits of the file.

Locating the Element Pointer

Since the element pointer can be moved randomly in a binary file, it is useful to have a method of finding out where the pointer is at any time during the execution of a program.

The LOC function will provide the location of the pointer in the file referenced.

For example:

```
LET X=LOC(file designator)
```

In this statement, X represents any numeric variable, and the file designator is as previously described.

Example:

```
100 FILES F1;F2
    .
    .
210 LET A1=LOC(1)
    .
    .
290 PRINT LOC(2)
```

The value of the element pointer into file F1 will be stored in A1 (line 210).

The value of the element pointer into file F2 will be printed (line 290).

File Scratching

The form of the binary scratch file statement is

```
SCRATCH : file designator
```

where the file designator is as previously described.

The SCRATCH statement cause the data element pointer to be repositioned so that transmission to the file starts at the beginning of the file.

Example:

```
100 FILES VFIL
115 SCRATCH:1
125 FOR I=1 TO 20
140 WRITE:1, INT(10*RND(X))
155 NEXT I
```
The **SCRATCH**: statement may be replaced with a **SET**: statement to accomplish the same purpose:

\[
115 \text{ SET}: 1, 1
\]

A file being read can be written without being scratched. This means that files can be modified during program execution.

Example:

\[
100 \text{ FILES RFIL} \\
110 \text{ LET } Z1 = 0 \\
120 \text{ LET } S1 = 26 \\
130 \text{ SET}: 1, S1 \\
140 \text{ READ}: 1, X \\
150 \text{ IF } X = 0 \text{ THEN } 180 \\
160 \text{ SET}: 1, \text{LOC}(1)-1 \\
170 \text{ WRITE}: 1, Z1 \\
180 \text{ LET } S1 = S1 + 26 \\
190 \text{ IF } S1 < \text{EOF}(1) \text{ THEN } 130 \\
200 \text{ END}
\]

This example reads every 26th data element and goes back and writes over that element with a zero, provided that the element is not already zero.

**File Restoring**

The form of the binary restore file statement is

\[
\text{RESTORE} : \text{file designator}
\]

where the file designator is as previously described.

The **RESTORE**: statement will move the element pointer to the beginning of the designated file.

Example:

\[
100 \text{ FILES VFIL} \\
110 \text{ SCRATCH}: 1 \\
120 \text{ FOR } J = 1 \text{ TO } 50 \\
130 \text{ WRITE}: 1, J; \text{SQR}(J) \\
140 \text{ NEXT } J \\
150 \text{ RESTORE}: 1 \\
180 \text{ READ}: 1, X
\]

This program writes 100 data elements into VFIL. Then the element pointer is moved to the beginning of the file (line 150), and the first data element is read into X.

A file being written can be read before it is restored. If line 150 in the last example is omitted, reading will begin where writing stopped, and the 101st data element will be read into X.

**End-of-File Test**

For binary files the end-of-file test is a test for the end of file space. The **IF END** statement will test whether an end-of-file-space condition was detected by the last **READ**: or **WRITE**: statement. When reading a partially filled binary file, the **READ**: statement should be executed...
only the number of times required to read the legitimate data in the file. The IF END test would allow such a file to be read until all the space in the file had been exhausted.

The form of the statement is

\[
\text{IF END : file designator THEN line number}
\]

If the last READ: statement or WRITE: statement encountered the end of file space, the program will go to the line number specified in the IF END: statement. Otherwise the next sequential statement is executed.

Example:

100 FILES F1
110 SCRATCH : 1
120 FOR J = 1 TO 1000
130 IF END : 1 THEN 160
140 WRITE : 1, J/2
150 NEXT J
160 END

An intrinsic function is also provided to test for the end-of-file condition. The function LEN will retrieve the length of the referenced file. For example:

\[
\text{LET } X = \text{LEN(file designator)}
\]

In this statement, X represents any numeric variable, and the file designator is as previously described.

The functions LOC and LEN can replace the IF END: statement in the previous sample program as follows:

130 IF LOC(1) = LEN(1) THEN 160

When the element pointer into file F1 reaches the end of the file, writing stops.

If a program continues reading or writing a file after the end-of-file-space condition has been detected, the error message END OF FILE SPACE will be printed, and the program will continue executing.

MATRICES

The matrix operation statements available in BASIC are among the most powerful and useful in the entire language.

Following is a list of matrix statements.

- **MAT READ A,B,C**
  - Read matrices A, B, and C, their dimensions having been previously specified. Data is read in row-wise sequence.

- **MAT PRINT A,B;C**
  - Print matrices A, B, and C, with A and C in the regular format, but B closely packed.

- **MAT C = A + B**
  - Add matrices A and B and store the result in matrix C.

- **MAT C = A - B**
  - Subtract matrix B from matrix A and store the result in matrix C.

- **MAT C = A*B**
  - Multiply matrix A by matrix B and store the result in matrix C.

- **MAT C = INV(A)**
  - Invert matrix A and store the result in matrix C.
MAT C = TRN(A)  
Transpose matrix A and store the result in matrix C.

MAT C = (K)*A  
Multiply matrix A by the value represented by K. K may be either a number or an expression, but in either case it must be enclosed in parentheses.

MAT C = CØN  
Set each element of matrix C to one. CØN means constant.

MAT C = ZER  
Set each element of matrix C to zero.

MAT C = IDN  
Set the diagonal elements of matrix C to one's and the non-diagonal elements to zeroes, yielding an identity matrix.

MAT READ and MAT PRINT

Using the MAT READ and MAT PRINT statements, you can read data into or print data from a matrix without having to reference each element of the matrix individually.

Examples:

100 MAT READ A, F, H, G
150 MAT PRINT C
175 MAT READ Z
190 MAT PRINT A, L

Information is read into a matrix using the DATA statement. The elements in the DATA statement are taken in row order, that is,

\[ A_{1,1}, A_{1,2}, \ldots, A_{1,m}, A_{2,1}, A_{2,2}, \ldots, A_{2,m}, \ldots, A_{n,m} \]

Information is read from DATA statements until the matrix array is completely filled. Partial matrices cannot be read or printed.

Example:

110 DIM L(2,3), M(2,2)
150 MAT READ L, M
160 LET L(2,2) = -2*L(2,2)
200 MAT PRINT L, M
500 DATA 1,2,3,4,5,6,3,-12,0,7

Line 110 defines L as a 2 by 3 matrix and M as a 2 by 2 matrix. The MAT READ statement reads in row order from the DATA statement at line 500. The matrix element \( L_{2,2} \) is recomputed at line 160. The two matrices are then printed to yield:

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & -10 & 6
\end{bmatrix}
\quad \text{and} \quad
\begin{bmatrix}
3 & -12 \\
0 & 7
\end{bmatrix}
\]

Matrix Addition, Subtraction, and Multiplication

You can add, subtract, and multiply matrices using the matrix arithmetic statements. The matrix dimensions must be conformable for each operation. If dimensions are not conformable, execution is stopped and you receive a dimension error message.

Matrix arithmetic statements may take the forms

MAT C = A + B  
MAT C = A - B  
MAT P = Q*R
Only one operation can be done in each statement.

Example:

Calculate \[ H_{a,a} = [E]_{a,a} - [K]_{a,a} + [A]_{a,3} \times [B]_{3,a} \]

\[ 612 \text{ MAT } H = A \times B \]
\[ 615 \text{ MAT } H = H + E \]
\[ 618 \text{ MAT } H = H - K \]

Scalar Multiplication

You can multiply a matrix by a scalar expression using a statement of the form

\[ \text{MAT } X = (\text{expression}) \times D \]

where \( X \) and \( D \) are matrices and the expression in parentheses is a scalar quantity. The parentheses are required to indicate scalar rather than matrix multiplication. Only one operation per statement is allowed.

Examples:

\[ 100 \text{ MAT } F = (2) \times G \]
\[ 150 \text{ MAT } Q = (2.33 + M) \times Q \]
\[ 750 \text{ MAT } B = (N) \times A \]

Identity Matrix

An identity matrix is defined by a statement of the form

\[ \text{MAT } B = \text{IDN} \quad \text{or} \quad \text{MAT } R = \text{IDN}(\text{expression}, \text{expression}) \]

In the first statement, matrix \( B \) is set up as an identity matrix. If \( B \) is not defined to be square, you will receive a dimension error message. In the second statement, the size of the identity matrix \( R \) is determined at execution time by the value of the expression enclosed in parentheses.

Examples:

\[ 190 \text{ MAT } A = \text{IDN} \]
\[ 100 \text{ MAT } V = \text{IDN}(2N + 1, 2N + 1) \]
\[ 120 \text{ MAT } B = \text{IDN}(Q, Q) \]
\[ 130 \text{ MAT } W = \text{IDN} \]
\[ 140 \text{ MAT } C = \text{IDN}(1, 1) \]

Matrix Transposition

Matrices are transposed using the form

\[ \text{MAT } Y = \text{TRN}(Z) \]

where \( Y \) and \( Z \) are both matrices. The matrix \( Z \) transpose will replace matrix \( Y \). \( Y \) and \( Z \) must conform. Matrix transposition in place (\( \text{MAT } A = \text{TRN}(A) \)) is not allowed.

Examples:

\[ 300 \text{ MAT } G = \text{TRN}(H) \]
\[ 400 \text{ MAT } U = \text{TRN}(V) \]
Matrix Inversion

Matrices are inverted using the form

\[
\text{MAT } I = \text{INV}(J)
\]

where \( I \) and \( J \) are both matrices. \( I \) will contain the matrix \( J \) inverse. \( I \) and \( J \) must conform. Matrix inversion in place (\( \text{MAT } A = \text{INV}(A) \)) is not allowed. If a matrix is singular, you will receive the message NEARLY SINGULAR MATRIX.

Examples:

```
500 \text{MAT } K = \text{INV}(L)
560 \text{MAT } A = \text{INV}(B)
```

Matrix ZER and CØN Functions

The ZER function is used to zero out all elements of a matrix. It may also be used to redefine the dimensions of a matrix during execution as described in "Dimensioning." As an example

\[
\text{MAT } C = \text{ZER}
\]

will zero out the elements of matrix \( C \).

The CØN function is used to set all elements of a matrix to one's. As an example

\[
\text{MAT } C = \text{CØN}
\]

will set all elements of matrix \( C \) to one's.

Dimensioning

Every matrix variable used in a program must be given a single-letter name.

A matrix variable must be defined in a DIM statement, which sets aside the amount of storage required by the matrix variable during execution of the program. For example:

```
\text{DIM } P(3,4), Q(5,5)
```

The DIM statement defines two matrices, \( P \) and \( Q \). \( P \) is defined as a 12 element matrix, and \( Q \) as a 25 element matrix. Note that the first element of \( P \) is \( P(1,1) \) and the last element \( P(3,4) \). The elements of \( Q \) run from \( Q(1,1) \) through \( Q(5,5) \). All matrix variables must be doubly dimensioned, as shown here.

Before any computation using the MAT statements, you must declare the precise dimensions of all matrices to be used in the computation. Four of the MAT statements are used for this purpose:

```
\text{MAT } \text{READ } C(M,N)
\text{MAT } C = \text{ZER}(M,N)
\text{MAT } C = \text{CØN}(M,N)
\text{MAT } C = \text{IDN}(N,N)
```

The first three statements specify matrix \( C \) as consisting of \( M \) rows and \( N \) columns. The fourth statement specifies matrix \( C \) as a square matrix of \( N \) rows and \( N \) columns.

These same statements may be used to redimension a matrix during running. A matrix may be redimensioned to either a larger or a smaller matrix, provided the new dimensions do not require more storage space than was originally reserved by the DIM statement. To illustrate, consider the following.
Example:

110 DIM A(8,8), B(8,8), C(8,8)
150 MAT READ A(2,2), B(2,2)
160 MAT C = ZER(2,2)
          ;
200 MAT A = IDN(8,8)
210 MAT READ B(4,4), C(4,4)

Note that the DIM statement reserves enough storage to accommodate three matrices, each consisting of 64 elements. The initial MAT READ specifies the dimensions of both matrices A and B as 2 rows and 2 columns.

The MAT READ also reads the number of values required by the dimensions into the storage that was reserved by the DIM statement. It reads them in row-wise sequence. In the initial MAT READ, the elements in the order read are A(1,1), A(1,2), A(2,1), A(2,2), B(1,1), B(1,2), B(2,1), and B(2,2). Statement 160 uses the ZER to specify dimensions and to zero the elements of matrix C. Statements 200 and 210 illustrate redimensioning. Matrix A is redimensioned as an 8 row, 8 column identity matrix; and matrices B and C are redimensioned as 4 row, 4 column matrices into which data is to be read.

The combination of ordinary BASIC statements and MAT statements makes BASIC very powerful, but you must be careful about dimensions. In addition to having both a DIM statement and a declaration of current dimension, you should watch your use of the MAT statements. For example, a matrix product $MATC = A*B$ may be illegal for either of two reasons: A and B may have such dimensions that the product is not defined, or C may have the wrong dimensions for the answer. In either case you will receive the "DIMENSI(>N ERR(>R message.

Examples

Two programs follow that illustrate some of the capabilities of the MAT statements. In the first program, the values for $M$ and $N$ are read. Using these two values as indices, statement 120 sets the dimensions for matrices $A, B, D, \text{and } G$. The values for the elements of these four matrices are read. Then, in sequence:

- The dimensions of matrix C are specified and the elements set to zero (line 130).
- Matrix A is printed (line 150).
- Matrix B is printed (line 170).
- The sum of matrices A and B is found and stored in C (line 180).
- Matrix C is printed (line 200).
- The dimensions for matrix F, a vector, are set and the elements set to zero (line 210).
- The product of matrices C and D is computed and stored in F (line 220).
- The dimensions for matrix H (single value) are specified and the elements set to zero (line 230).
- Finally, the product of matrices G and F is found and stored in H and printed (lines 240, 260).

In the second program, a value $N$ is read that determines the order of the Hilbert matrix segment to be computed, stored, and printed. Next the matrix is inverted and printed. Finally the Hilbert matrix is multiplied by its own inverse, and the resulting product matrix is printed. Notice that line 290 specifies $N$ as equal to 2 to produce the first three matrices of order 2, and later returns to read in the data "8", redimensions to a larger array--larger than 2, but smaller than the original 20--and produces more output.
MATRIX PROGRAM EXAMPLE 1:

```
100 DIM A(5,5),B(5,5),C(5,5),D(5,5),E(5,5),F(5,5),G(5,5),H(5,5)
110 READ M,N
120 MAT READ A(M,M),B(M,M),D(M,N),G(N,M)
130 MAT C=ZER(M,M)
140 PRINT "MATRIX A OF ORDER";M
150 MAT PRINT A;
160 PRINT "MATRIX B OF ORDER";M
170 MAT PRINT B;
180 MAT C=A+B
190 PRINT "C=A+B"
200 MAT PRINT C;
210 MAT F=ZER(M,N)
220 MAT H=ZER(N,N)
230 MAT H=G*F
240 PRINT "H"
250 MAT PRINT H;
260 DATA 3,1
270 DATA 1,2,3,4,5,6,7,8,9,9,8,7,6,5,4,3,2,1,1,2,3,3,2,1
280 DATA 1,2,3,4,5,6,7,8,9,9,8,7,6,5,4,3,2,1,1,2,3,3,2,1
999 END
RUN

MAT= 15:44
MATRIX A OF ORDER 3
 1 2 3
 4 5 6
 7 8 9

MATRIX B OF ORDER 3
 9 8 7
 6 5 4
 3 2 1

C=A+B
 10 10 10
 10 10 10
 10 10 10

H
360
```
MATRIX PROGRAM EXAMPLE 2:

```
100 DIM A(20,20),B(20,20),C(20,20)
110 READ N
120 MAT A=CON(N,N)
130 MAT B=CON(N,N)
140 MAT C=ZER(N,N)
150 FOR I=1 TO N
160 FOR J=1 TO N
170 LET A(I,J)=1/(I+J-1)
180 NEXT J
190 NEXT I
200 PRINT "HILBERT MATRIX OF ORDER";N
210 MAT PRINT A;
220 MAT B=INV(A)
230 PRINT "INVERSE OF HILBERT MATRIX OF ORDER";N
240 MAT PRINT B;
250 MAT C=A*B
260 PRINT "HILBERT MATRIX TIMES ITS OWN INVERSE ORDER";N
270 MAT PRINT C;
280 GOTO 110
290 DATA 2,3
999 END
```

RUN

MAT-2 15146

HILBERT MATRIX OF ORDER 2

```
1 -5
-5 333333
```

INVERSE OF HILBERT MATRIX OF ORDER 2

```
4 -6
-6 12
```

HILBERT MATRIX TIMES ITS OWN INVERSE ORDER 2

```
1 0
-3.72529E-09 1
```

HILBERT MATRIX OF ORDER 3

```
1 -5 -333333
-5 333333 25
-333333 25 2
```

INVERSE OF HILBERT MATRIX OF ORDER 3

```
9 -36 30
-36 192 -180
30 -180 180
```

HILBERT MATRIX TIMES ITS OWN INVERSE ORDER 3

```
1 -1.78814E-07 0
-2.23517E-08 1 -5.96046E-08
-1.49012E-08 -1.78814E-07 1
```

OUT OF DATA IN 110
EXAMPLES OF ADVANCED BASIC PROGRAMS

Following are two sample programs illustrating the use of many of the advanced capabilities of BASIC. The first program is developed in an inventory case problem, and makes use of a BCD file. The second program uses a binary file to store personnel information.

Inventory Problem

Mr. Swift, a storekeeper, would like to know how any five items in his store are selling in any given month. He would like a permanent file of the items that were sold each week over a four week period. He wants to update his file at the end of each week, and he may or may not want to get a complete written record of his sales. He may want to get a written report at any time during the month.

The record should consist of an easy to read table listing the items and the number sold in each week. The table should also show the total number of items for each week, the total number of each item sold to date, and the total number of all items sold to date.

The five items that Mr. Swift would like to check are salt, pepper, sugar, nutmeg, and coffee. The month is March.

The following program results from Mr. Swift’s requirements. The program is explained by remarks included in it.

```
100 FILES STOCK
110 DIM A(4,3)
120 FOR I=0 TO 5
130 READ AS(I)
140 NEXT I
150
160 REM W=INITIAL PASS FLAG, P=PRINTOUT FLAG
170 REM W=0 INITIAL PASS, P=0 PRINTOUT DESIRED
180 READ W,P
190 IF W<0 THEN 300
200
210 REM FOR THE INITIAL PASS, WRITE THREE ZER0ES. THERE IS NO
220 REM DATA INITIALLY, AND SOMETHING MUST BE WRITTEN BEFORE
230 REM IT CAN BE READ.
240 SCRATCH#1
250 WRITE#1,0,0,0,0
260 RESTORE#1
270
280 REM READ IN DATA WRITTEN INTO THE FILE FROM PREVIOUS WEEKS.
290 REM X---ITEM, Y---WEEK, A(X,Y)---NUMBERS OF ITEMS SOLD TO DATE.
300 READ#1,X,Y,A(X,Y)
310 IF END#1 THEN 350
320 G0 TO 300
330
340 REM READ DATA FOR THIS WEEK OR DATA MISSED FROM PREVIOUS WEEKS.
350 READ X,Y,Z
360 IF X<0 THEN 430
370
380 REM WHEN X IS NEGATIVE, THE DATA READ IS FINISHED.
390 LET A(X,Y)=A(X,Y)+Z
400 G0 TO 350
410
420 REM WRITE THE UPDATED INFO TO THE PERMANENT DATA FILE.
430 SCRATCH#1
440 FOR X=0 TO 4
450 FOR Y=0 TO 3
460 WRITE#1,X,Y;A(X,Y)
470 NEXT Y
480 NEXT X
490
500 REM IS A PRINTOUT WANTED? 0---YES  -1---NO
510 IF P<0 THEN 870
```
Suppose that two weeks have passed and Mr. Swift wants a record of his sales to date. He runs the program with data in line 1000 as shown above. The -1 specifies not the first week, and the 0 specifies a printout of the file data. Also he enters data in line 1010 as shown above. The -1 indicates the termination of data, and the two final zeroes are dummy data put in to satisfy the READ. He gets the following results.

RUN

SALT 8:39
MARCH

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>TOTALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALT</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>PEPPER</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>SUGAR</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>NUTMEG</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>COFFEE</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>52</td>
</tr>
</tbody>
</table>
At the end of the third week, Mr. Swift wants to make an update. He wants a printout. All he must do is replace line number 1010, as shown below. The BCD data file is updated, and the printout shows the entries for the third week and the resulting changes in the totals.

```
1010 DATA 0,2,7, 1,2,5, 2,2,1, 3,2,5, 4,2,12, -1,0,0
RUN
```

**SALT**  14:00

**MARCH**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>TOTALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALT</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>PEPPER</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>SUGAR</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>NUTMEG</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>COFFEE</td>
<td>2</td>
<td>9</td>
<td>12</td>
<td>0</td>
<td>23</td>
</tr>
</tbody>
</table>

24  28  30  0  82

Personnel Information

Following is a typical example of the use of binary files. There are two programs. **WØRKER** establishes the data base in the file INFØ. **WØRK1** shows how values can be altered at will.

```
100 FILES INFØ
110 SCRATCH t1
120 READ A,S,A1,D,S1,V
130 WRITE1,A,S,A1,D,S1,V
140 G0 TO 120
150 DATAGORDON,9345,27,0,4,0
160 DATAPLUMMER,10200,30,4,0.5
170 DATAGARANTINE,8600,22,0,2,7
180 DATATHOMAS,11550,29,3,0,2
190 DATACHENEY,8800,25,0,4,6

RUN
WØRKER  14:06
OUT OF DATA IN 120

100 FILES INFØ
110 READ A,B,C
120 G0SUB 200
130 SET1,A*B+B
140 READ1,X
150 LET X=X+C
160 SET1,L0C(1)=1
170 WRITE1,X
180 G0 SUB 240
190 STØP
200 PRINTUSING 220
210 PRINT
220:NAME SALARY AGE DEPENDENTS SICK DAYS VACATION DAYS
230:"S"### #### ### ####
240 SET1,A*B+1
250 READ1,A,S,A1,D,S1,V
260 PRINTUSING 230,A,S,A1,D,S1,V
270 RETURN
280
```
290 REM DATA FORMAT --- A-NAME, B-CODE VALUE, C-AMOUNT TO BE ADDED
300 REM NAMES---- 0-GORDON, 1-PLUMMER, 2-GARANTINO, 3-THOMAS
310 REM 4-CHENEY
320
330 REM CODE------ 1-NAME, 4-SALARY, 5-AGE, 6-DEPENDENTS, 7-SICK DAYS
340 REM 8-VACATION DAYS
350
360 REM FOR DATA --- 2,4,820 --- ALTER MR. GARANTINO'S SALARY BY $820
370
380 DATA 2,4,820

RUN

W0RK1 14:11

NAME   SALARY   AGE   DEPENDENTS   SICK DAYS   VACATION DAYS
GARANTINO $ 8600 22   0         2          7
GARANTINO $ 9420 22   0         2          7

The second line of the printout shows the $820 increase in Mr. Garantino's salary.
Appendix A  Error Messages

Because most programs under development contain errors, a series of error messages is included in BASIC. Some of the messages are received during compilation and others during execution of a program. Many of the messages not only identify the type of error, but indicate the line number where the error occurred. In the following table, XXX means a line number.

During execution, some messages occur that do not stop execution, but inform you of irregular conditions existing in identified lines of your program. Other messages, however, point out more serious errors that cause execution to stop.

Compilation Errors

<table>
<thead>
<tr>
<th>MESSAGE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUT PRØGRAM ÖR DIMS</td>
<td>Either the program is too long, or the amount of space reserved by the DIM statements is too much, or both. Cut the length of the program, reduce the size of the lists and tables, reduce the length of printed labels, or reduce the number of simple variables.</td>
</tr>
<tr>
<td>DIMENSION TØØ LARGE IN XXX</td>
<td>The size of a list or table is too large. Make it smaller. Maximum dimension is A(1022).</td>
</tr>
<tr>
<td>FILE NØT DEFINED IN XXX</td>
<td>The file reference statement is missing.</td>
</tr>
<tr>
<td>FILES NØT FIRST IN XXX</td>
<td>The file reference statement is preceded by an executable statement.</td>
</tr>
<tr>
<td>FÔR WİTHÖUT NEXT</td>
<td>A NEXT statement is missing. This message can occur in conjunction with NEXT WİTHÖUT FÔR.</td>
</tr>
<tr>
<td>ILLEGAL CÔNSTANT IN XXX</td>
<td>A number is out of bounds(&gt; 5.78960E76).</td>
</tr>
<tr>
<td>ILLEGAL FÔRMULA IN XXX</td>
<td>Perhaps the most common error message. May indicate missing parentheses, illegal variable names, missing multiply signs, illegal numbers, or other errors. Check the statement thoroughly.</td>
</tr>
<tr>
<td>ILLEGAL INSTROKTİON IN XXX</td>
<td>Other than one of the 26 legal BASIC instructions has been used following the line number.</td>
</tr>
<tr>
<td>ILLEGAL NUMBER IN XXX</td>
<td>The line number is of incorrect form or contains more than 5 digits.</td>
</tr>
<tr>
<td>ILLEGAL PRÔGRAM NAME IN XXX</td>
<td>A program name in a CALL or CHAIN statement is a name with more than 6 characters and is other than a library program name. A program cannot call itself.</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>MEANING</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ILLEGAL RELATION IN XXX</td>
<td>Something is wrong with the relational expression in an IF--THEN statement. Check to see if you have used one of the 6 permissible relational symbols.</td>
</tr>
<tr>
<td>ILLEGAL VARIABLE IN XXX</td>
<td>An illegal variable name has been used.</td>
</tr>
<tr>
<td>IMAGE TABLE OVERFLOW</td>
<td>More than 100 image statements have been included in the program.</td>
</tr>
<tr>
<td>INCORRECT FORMAT IN XXX</td>
<td>The format of the statement is wrong.</td>
</tr>
<tr>
<td>NEXT WITHOUT FOR IN XXX</td>
<td>There is an incorrect NEXT statement, perhaps with a wrong variable given. Check also for incorrectly nested FOR statements.</td>
</tr>
<tr>
<td>NO DATA</td>
<td>There is at least one READ statement in the program, but there are no DATA statements.</td>
</tr>
<tr>
<td>OVER 10 SUBROUTINES</td>
<td>The program tried to call more than 10 different programs.</td>
</tr>
<tr>
<td>PROGRAM NOT SAVED (Program Name)</td>
<td>A program named in a CALL statement is not saved.</td>
</tr>
<tr>
<td>PROGRAM TOO LONG IN XXX</td>
<td>The program is too long for the available storage. Cut the size of the program or dimensions.</td>
</tr>
<tr>
<td>PROGRAM WON'T FIT (Program Name)</td>
<td>The program called is too large to fit in the remaining available space.</td>
</tr>
<tr>
<td>REDIMENSIONED ARRAY IN XXX</td>
<td>An array has previously been dimensioned.</td>
</tr>
<tr>
<td>STRING TOO LONG IN XXX</td>
<td>A string has more than 15 characters.</td>
</tr>
<tr>
<td>TOO MANY CONSTANTS IN XXX</td>
<td>A statement contains constants that result in more than 75 different constants in the program. Ex-ample: LET A(4) = 1.24; 4 and 1.24 are constants.</td>
</tr>
<tr>
<td>TOO MANY FILES</td>
<td>The number of files referenced caused the program size limit to be exceeded. (At least 8 files will always be allowed.)</td>
</tr>
<tr>
<td>TOO MANY GOTOS IN XXX</td>
<td>More than 79 different line number references were made in GOTO, IF--THEN, GOSUB, or ON--GOTO statements.</td>
</tr>
<tr>
<td>TOO MANY LOOPS</td>
<td>There are more than 26 FOR--NEXT combinations in the program.</td>
</tr>
<tr>
<td>TOO MUCH DATA</td>
<td>The program contains more than 1280 numbers, or too many numbers and strings, or too many strings as data.</td>
</tr>
<tr>
<td>UNDEFINED FUNCTION</td>
<td>A function such as FNF( ) has been used without appearing in a DEF statement. Check for typographical errors.</td>
</tr>
</tbody>
</table>
MESSAGE

UNDEFINED IMAGE XXX

UNDEFINED NUMBER IN XXX

Execution Errors--Execution Continued

MESSAGE

ABSOLUTE VALUE RAISED TO POWER IN XXX

ATTEMPT TO READ A NUMBER AS A STRING VARIABLE

DIVISION BY ZERO IN XXX

END OF FILE IN XXX

END OF FILE SPACE IN XXX

INPUT DATA NOT IN CORRECT FORMAT, RETYPE IT

LOG OF NEGATIVE NUMBER IN XXX

LOG OF ZERO IN XXX

OVERFLOW IN XXX

SQUARE ROOT OF A NEGATIVE NUMBER IN XXX

UNDERFLOW IN XXX

MEANING

A PRINT USING statement references line XXX, but line XXX either does not exist or is not an image statement.

The statement number appearing in a GOTO, IF--THEN, GOSUB, or ON--GOTO statement does not appear as line number in the program.

A computation of the form \((-3)^2.7\) has been attempted. The computer supplies \(\text{ABS}(-3)^2.7\) and continues. Note: \((-3)^3\) is correctly computed to give \(-27\).

Self explanatory.

A division by zero has been attempted. The computer supplies \(+\infty\) (about 5.78960E76) and continues running the program.

An attempt has been made to read data from a BCD file after all data has been read. The file pointer is located at the end of the file. No data is transmitted.

After all physical space in a file has been used, an attempt has been made to write into a BCD file, or an attempt has been made to read from or write into a binary file. No data is transmitted.

Self explanatory.

The program has attempted to calculate the logarithm of a negative number. The computer supplies the logarithm of the absolute value and continues.

The program has attempted to calculate the logarithm of zero. The computer supplies \(-5.78960E76\) and continues.

A number larger than about 5.78960E76 has been generated. The computer supplies \(\pm 5.78960E76\) and continues running the program.

The program has attempted to extract the square root of a negative number. The computer supplies the square root of the absolute value and continues.

A number smaller in absolute size than about 4.31809E-78 has been generated. The computer supplies zero and continues. In many circumstances, underflow is permissible and may be ignored.
<table>
<thead>
<tr>
<th>MESSAGE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZERØ TÔ A NEGATIVE POWER IN XXX</td>
<td>A computation of the form (0^t (-1)) has been attempted. The computer supplies (+m) (about (5.78960E76)) and continues.</td>
</tr>
<tr>
<td>Execution Errors--Execution Terminated</td>
<td></td>
</tr>
<tr>
<td>BAD IMAGE IN XXX</td>
<td>There are syntax errors in the image statement referenced by line number XXX, or an attempt has been made to put numeric data in an alphanumeric field, or alphanumeric data in a numeric field.</td>
</tr>
<tr>
<td>CALLS ÔR GÕSUB NESTED TÔØ DEEPLY IN XXX</td>
<td>Too many CALLs or GÕSUBs without a RETURN. It may be that subroutines are being left by GÕTÔ or IF--THEN statements rather than by RETURNs. The program stops.</td>
</tr>
<tr>
<td>DATA FILE (LINE XXX) FÕORMAT ERRÔR</td>
<td>At line XXX of the data file being read, data is not in the required format.</td>
</tr>
<tr>
<td>DIMENSION ERROR IN XXX</td>
<td>A dimension inconsistency has occurred in connection with one of the MAT statements. The program stops.</td>
</tr>
<tr>
<td>EXPRESSION ØUT ØF RANGE</td>
<td>The range of an ÔN--GÔTÔ statement is incorrect. Example: ÔN X GÔTÔ 10,20,30. When the integer value of X is either minus, zero, or greater than 3, the expression is out of range.</td>
</tr>
<tr>
<td>FILE NÔT BCD</td>
<td>An attempt has been made to do a BCD file read or write on a binary file.</td>
</tr>
<tr>
<td>FILE NÔT BINARY</td>
<td>An attempt has been made to do a binary file read or write on a BCD file.</td>
</tr>
<tr>
<td>FILE(S) NÔT SAVED: (File Name)</td>
<td>The files indicated have been referenced but are not saved in your library.</td>
</tr>
<tr>
<td>ILLEGAL FILE DESIGNATÔR IN XXX</td>
<td>The file designator is less than unity, non-integral, or greater than the number of referenced files.</td>
</tr>
<tr>
<td>ILLEGAL POINTÈR</td>
<td>The element pointer is less than unity, non-integral, or greater than the length of the referenced file.</td>
</tr>
<tr>
<td>NEARLY SINGULAR MATRIX IN XXX</td>
<td>The INV operation in MAT has encountered a matrix with zero or nearly zeropivotal elements. The matrix being inverted is singular or nearly so. Note, however, that this check is not 100 percent reliable. For instance, this message need not occur even if the inverse is meaningless, as with high order Hilbert matrices. If this error occurs, the program stops.</td>
</tr>
<tr>
<td>ØUT ØF DATA IN XXX</td>
<td>A READ statement for which there is no DATA has been encountered. If this means a normal end of your program, ignore the message. Otherwise, it means that you haven’t supplied enough DATA. In either case, the program stops.</td>
</tr>
</tbody>
</table>
MESSAGE | MEANING
--- | ---
READING BCD IN XXX | An attempt has been made to write into a BCD read mode file. Indicates a logic error or no SCRATCH statement encountered before read mode activity.
RETURN BEFORE GOSUB OR CALL IN XXX | A RETURN has been encountered before the first GOSUB or CALL in the program. Note: BASIC does not require the GOSUB to have an earlier statement number—only to execute a GOSUB before executing a RETURN. The program stops.
SUBSCRIPT ERROR IN XXX | A subscript has been called for that lies outside the range specified in the DIM statement, or, if no DIM statement applies, outside the range 0 through 10. The program stops.
WRITING BCD IN XXX | An attempt has been made to read or backspace a BCD write mode file. Indicates a logic error or no RESTORE statement encountered before read mode activity.
Appendix B  Limitations on BASIC

There are some limitations imposed on BASIC by the limited amount of computer storage. Listed below are some of these limitations, in particular, those that are related to the error messages in Appendix A. The reader should realize that although the BASIC language itself is fixed, in time some of these limitations may be relaxed slightly.

<table>
<thead>
<tr>
<th>ITEM</th>
<th>LIMITATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source program size</td>
<td>The source program may not consist of more than 256 lines. It may not contain more than 6144 characters.</td>
</tr>
<tr>
<td>Constants</td>
<td>The total number of different constants must not exceed 75.</td>
</tr>
<tr>
<td>Data</td>
<td>There can be no more than 1280 data numbers.</td>
</tr>
<tr>
<td>FOR statements</td>
<td>There can be no more than 26 FOR statements in a program.</td>
</tr>
<tr>
<td>GOTO T0, IF--THEN, GOSUB, and ON--G0 T0 statements</td>
<td>The total number of different references in these statements cannot exceed 79.</td>
</tr>
<tr>
<td>Compiled program size</td>
<td>Cannot exceed 4148 words.</td>
</tr>
<tr>
<td>Image statements</td>
<td>Maximum of 100.</td>
</tr>
<tr>
<td>Dimension of array</td>
<td>A singly dimensioned array cannot exceed 1022. The limitations on a doubly dimensioned array with dimensions X, Y, are: (1) X cannot exceed 1022, (2) Y cannot exceed 510, and (3) the product of X+1 and Y+1 cannot exceed 2074.</td>
</tr>
<tr>
<td>Naming of variables</td>
<td>The variable A is distinct from the element A(0).</td>
</tr>
<tr>
<td>Subscripting</td>
<td>Numeric variable names consisting of two characters may not be subscripted.</td>
</tr>
</tbody>
</table>
Appendix C  Comparison Order for BASIC Characters

BASIC characters are compared in their BASIC code representations. The following table gives the BASIC code number for each character. Codes of nonprinting characters are enclosed in parentheses.

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>24</td>
<td>D</td>
<td>53</td>
<td>$</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>25</td>
<td>E</td>
<td>54</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(55)</td>
<td></td>
<td>End of Message</td>
</tr>
<tr>
<td>02</td>
<td>2</td>
<td>26</td>
<td>F</td>
<td>56</td>
<td>&gt;</td>
</tr>
<tr>
<td>03</td>
<td>3</td>
<td>27</td>
<td>G</td>
<td>57</td>
<td>′</td>
</tr>
<tr>
<td>04</td>
<td>4</td>
<td>30</td>
<td>H</td>
<td>60</td>
<td>(space)</td>
</tr>
<tr>
<td>05</td>
<td>5</td>
<td>31</td>
<td>I</td>
<td>61</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(32) Bell</td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>6</td>
<td>33</td>
<td>. (period)</td>
<td>62</td>
<td>S</td>
</tr>
<tr>
<td>07</td>
<td>7</td>
<td>34</td>
<td>&quot; (quote)</td>
<td>63</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>35</td>
<td>?</td>
<td>64</td>
<td>U</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>36</td>
<td>&lt;</td>
<td>65</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(37) Carriage Ret.</td>
<td>66</td>
<td>W</td>
</tr>
<tr>
<td>12</td>
<td>' (apostrophe)</td>
<td>40</td>
<td>- (minus)</td>
<td>67</td>
<td>X</td>
</tr>
<tr>
<td>13</td>
<td>:</td>
<td>41</td>
<td>J</td>
<td>70</td>
<td>Y</td>
</tr>
<tr>
<td>14</td>
<td>(</td>
<td>42</td>
<td>K</td>
<td>71</td>
<td>Z</td>
</tr>
<tr>
<td>15</td>
<td>;</td>
<td>43</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(72) Line Feed</td>
<td>73</td>
<td>, (comma)</td>
</tr>
<tr>
<td>16</td>
<td>=</td>
<td>44</td>
<td>M</td>
<td>74</td>
<td>)</td>
</tr>
<tr>
<td>17</td>
<td>\</td>
<td>45</td>
<td>N</td>
<td>75</td>
<td>[</td>
</tr>
<tr>
<td>20</td>
<td>+</td>
<td>46</td>
<td>O</td>
<td>76</td>
<td>]</td>
</tr>
<tr>
<td>21</td>
<td>A</td>
<td>47</td>
<td>P</td>
<td>(77) Fill</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>B</td>
<td>50</td>
<td>Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>C</td>
<td>51</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(52) Tab</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix D  Using the Time-Sharing System

The Mark I Time-Sharing System consists of a GE-235 computer with a number of input-output stations, currently Model 33 and Model 35 Teletypes. Those using the input-output stations are able to share the use of the computer with each other so as to suggest that each one has sole use of the computer. The teletypewriters are the devices through which the user communicates with the computer. This appendix contains elementary instructions for using the Time-Sharing System. For complete information, see the Mark I Time-Sharing Service Command System Reference Manual (229116).

The Keyboard

The teletypewriter keyboard is a standard typewriter keyboard for the most part. There are three special keys the user must be familiar with.

RETURN  The RETURN key is located at the right-hand end of the third row of keys, and does more than act as an ordinary carriage return. The computer ignores the line being typed until this key is pushed.

CTRL  The CTRL (control) key is located at the left-hand end of the third row of keys. When it is pressed along with the X key, the computer deletes the entire line being typed. This also acts as a carriage return.

The backwards arrow key is the shift of $\theta$. It is used to delete the character or space immediately preceding the -. If this key is pressed N times, the N preceding characters or spaces will be deleted.

Examples:

ABCWT--DE appears as ABCDE when RETURN is pushed.

AB C--CDE appears as ABCDE when RETURN is pushed.

Some languages available on the Time-Sharing System use the three characters $\\$, $\lbrack$, and $\rbrack$. They are located on the keys L, K, and M, respectively, when either SHIFT key is pushed.

Teletypewriter Operation

Besides the keyboard itself there are four control buttons necessary to operate the machine.

<table>
<thead>
<tr>
<th>Button</th>
<th>Location</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ØRIG</td>
<td>Leftmost of six small buttons on the right.</td>
<td>Turns on the teletypewriter and connects it to the phone line.</td>
</tr>
<tr>
<td>CLR</td>
<td>Next to ØRIG.</td>
<td>Turns off the teletypewriter and disconnects the phone circuit.</td>
</tr>
<tr>
<td>LOC LF</td>
<td>Left of the space bar on Model 35 Teletypes only.</td>
<td>Feeds paper to permit tearing it off.</td>
</tr>
</tbody>
</table>
BUZ-RLS Rightmost of six small buttons on the right. Turns off the buzzer that signals a low paper supply.

If the teletypewriter is on a direct line to the computer, pushing the ORIG button is all that is necessary to connect up with the computer. To disconnect from the computer, type GÔÔDBYE or BYE. If that fails, push CLR.

In order to connect with the computer from a teletypewriter not on a direct line:

- Push the ORIG button and wait for the dial tone.
- Dial one of the numbers at the Time-Sharing Center.

In order to disconnect, type GÔÔDBYE or BYE, and if that fails, push CLR.

Control Commands

There are a number of commands that may be given to the computer by typing the command at the start of a new line, with no line number, and following the command with a carriage return. The following table lists some of the most frequently used of these commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATALOG</td>
<td>The computer types a list of the names of all the programs currently saved under that user number.</td>
</tr>
<tr>
<td>EDIT</td>
<td>Gives a brief explanation of the format used in the EDIT commands.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>Gives you an idea of the length of the program, to the nearest 200 characters. The maximum length of one program is 6400 characters.</td>
</tr>
<tr>
<td>LIST</td>
<td>Causes an up-to-date listing of the program to be typed out.</td>
</tr>
<tr>
<td>LIST--XXXXX</td>
<td>Causes an up-to-date listing of the program to be typed out beginning at line number XXXXX and continuing to the end.</td>
</tr>
<tr>
<td>NEW</td>
<td>Erases from working storage the program currently being worked on, and asks for a NEW FILE NAME.</td>
</tr>
<tr>
<td>ÔLD</td>
<td>Erases from working storage the program currently being worked on, and asks for an ÔLD FILE NAME.</td>
</tr>
<tr>
<td>RENAME</td>
<td>Permits you to change the name of the program you are currently working on, but does not destroy the program.</td>
</tr>
<tr>
<td>RUN</td>
<td>Begins the computation of a program.</td>
</tr>
<tr>
<td>RUN (typed during computation)</td>
<td>Gives an indication that a program is running and how much machine time has elapsed since the run began.</td>
</tr>
<tr>
<td>SAVE</td>
<td>Saves the program intact for later use. To retrieve a saved program, type ÔLD.</td>
</tr>
<tr>
<td>SCRATCH</td>
<td>Destroys the program currently being worked on, but leaves the user number and program name intact. It gives you a clean sheet to work on.</td>
</tr>
<tr>
<td>STATUS</td>
<td>Gives an indication of the status of the teletypewriter you are using (running, idle, or disconnected).</td>
</tr>
<tr>
<td>STÔP</td>
<td>Stops the computation at once. It can be typed only when the teletypewriter is not printing.</td>
</tr>
<tr>
<td>Command</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>Permits you to change systems (BASIC, ALGOL, etc.) without going through the sign-on sequence again.</td>
</tr>
<tr>
<td>TTY</td>
<td>Supplies the following information: teletypewriter number, user number, language being used, program being used, and status of teletypewriter.</td>
</tr>
<tr>
<td>UNSAVE</td>
<td>Erases a saved program from memory. Since the memory of the computer is finite, this command should be used to free space in storage for other users' programs.</td>
</tr>
<tr>
<td>Topic</td>
<td>Page(s)</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Acronym BASIC explained</td>
<td>1</td>
</tr>
<tr>
<td>Advanced BASIC</td>
<td>19-58</td>
</tr>
<tr>
<td>Advanced BASIC statements</td>
<td>31-37</td>
</tr>
<tr>
<td>Alphanumeric data</td>
<td></td>
</tr>
<tr>
<td>Definition of string</td>
<td>19</td>
</tr>
<tr>
<td>Dimensioning</td>
<td>19</td>
</tr>
<tr>
<td>IF--THEN statements</td>
<td>21</td>
</tr>
<tr>
<td>INPUT statements</td>
<td>20</td>
</tr>
<tr>
<td>LET statements</td>
<td>19, 20</td>
</tr>
<tr>
<td>PRINT statements</td>
<td>21</td>
</tr>
<tr>
<td>READ and DATA statements</td>
<td>20</td>
</tr>
<tr>
<td>String size</td>
<td>19</td>
</tr>
<tr>
<td>String variable</td>
<td>19</td>
</tr>
<tr>
<td>Arithmetic operations</td>
<td>5</td>
</tr>
<tr>
<td>BASIC acronym explained</td>
<td>1</td>
</tr>
<tr>
<td>BCD files</td>
<td>39-43</td>
</tr>
<tr>
<td>Backspacing</td>
<td>43</td>
</tr>
<tr>
<td>Data separator option</td>
<td>39</td>
</tr>
<tr>
<td>End-of-file</td>
<td>41</td>
</tr>
<tr>
<td>End-of-file test</td>
<td>41, 42</td>
</tr>
<tr>
<td>End-of-space</td>
<td>41</td>
</tr>
<tr>
<td>Mode</td>
<td>39</td>
</tr>
<tr>
<td>Reading</td>
<td>39, 40</td>
</tr>
<tr>
<td>Restoring</td>
<td>42</td>
</tr>
<tr>
<td>Scratching</td>
<td>42, 43</td>
</tr>
<tr>
<td>Writing</td>
<td>40, 41</td>
</tr>
<tr>
<td>Binary files</td>
<td>43-48</td>
</tr>
<tr>
<td>Block diagram</td>
<td>44</td>
</tr>
<tr>
<td>End-of-file test</td>
<td>47, 48</td>
</tr>
<tr>
<td>Locating element pointer</td>
<td>46</td>
</tr>
<tr>
<td>Random accessing</td>
<td>45, 46</td>
</tr>
<tr>
<td>Restoring</td>
<td>47</td>
</tr>
<tr>
<td>Scratching</td>
<td>46, 47</td>
</tr>
<tr>
<td>Writing</td>
<td>43, 44</td>
</tr>
<tr>
<td>CALL statement</td>
<td>33-35</td>
</tr>
<tr>
<td>CHAIN statement</td>
<td>36, 37</td>
</tr>
<tr>
<td>Commands</td>
<td>67, 68</td>
</tr>
<tr>
<td>Comparison order</td>
<td>65</td>
</tr>
<tr>
<td>Computed GÔ TÔ</td>
<td>16, 17</td>
</tr>
<tr>
<td>Control commands</td>
<td>67, 68</td>
</tr>
<tr>
<td>Data files <em>see also</em> BCD files, Binary files</td>
<td></td>
</tr>
<tr>
<td>BCD files defined</td>
<td>37</td>
</tr>
<tr>
<td>Binary files defined</td>
<td>37</td>
</tr>
<tr>
<td>Dummy catalog files</td>
<td>38, 39</td>
</tr>
<tr>
<td>File designator</td>
<td>38</td>
</tr>
<tr>
<td>File reference</td>
<td>37, 38</td>
</tr>
<tr>
<td>Naming</td>
<td>38</td>
</tr>
<tr>
<td>Saving space for</td>
<td>37-39</td>
</tr>
<tr>
<td>Data statement</td>
<td>3, 4, 14</td>
</tr>
<tr>
<td>Debugging *see Errors and debugging</td>
<td></td>
</tr>
<tr>
<td>Definitions</td>
<td></td>
</tr>
<tr>
<td>Line number</td>
<td>2</td>
</tr>
<tr>
<td>Program</td>
<td>1</td>
</tr>
<tr>
<td>Statement</td>
<td>2</td>
</tr>
<tr>
<td>DIM statement</td>
<td>10, 18</td>
</tr>
<tr>
<td>Dimensioning</td>
<td></td>
</tr>
<tr>
<td>Lists and tables</td>
<td>9, 10</td>
</tr>
<tr>
<td>Matrices</td>
<td>51, 52</td>
</tr>
<tr>
<td>Division by zero</td>
<td>4</td>
</tr>
<tr>
<td>E notation</td>
<td>6</td>
</tr>
<tr>
<td>Elementary BASIC statements</td>
<td>14-18</td>
</tr>
<tr>
<td>END statement</td>
<td>18</td>
</tr>
<tr>
<td>Entering data</td>
<td>3, 4, 14, 15</td>
</tr>
<tr>
<td>Error messages, table</td>
<td>59-63</td>
</tr>
<tr>
<td>Errors and debugging</td>
<td>11-14</td>
</tr>
<tr>
<td>Files <em>see</em> BCD files, Binary files, Data files</td>
<td></td>
</tr>
<tr>
<td>FOR and NEXT statements</td>
<td>17, 18</td>
</tr>
<tr>
<td>Format control characters</td>
<td>24</td>
</tr>
<tr>
<td>Formulas</td>
<td>5</td>
</tr>
<tr>
<td>Functions <em>see also</em> Mathematical functions</td>
<td></td>
</tr>
<tr>
<td>CLK</td>
<td>31</td>
</tr>
<tr>
<td>DEF</td>
<td>31</td>
</tr>
<tr>
<td>INT</td>
<td>28</td>
</tr>
<tr>
<td>RND</td>
<td>28-30</td>
</tr>
<tr>
<td>SGN</td>
<td>30</td>
</tr>
<tr>
<td>TIM</td>
<td>31</td>
</tr>
<tr>
<td>GÔ TÔ statement</td>
<td>16</td>
</tr>
<tr>
<td>GÔSUB and RETURN statements</td>
<td>32, 33</td>
</tr>
<tr>
<td>IF--THEN statement</td>
<td>3, 17</td>
</tr>
<tr>
<td>Image statement</td>
<td>24-28</td>
</tr>
<tr>
<td>INPUT statement</td>
<td>15</td>
</tr>
<tr>
<td>LET statement</td>
<td>3, 14</td>
</tr>
<tr>
<td>Line numbers</td>
<td>2, 4</td>
</tr>
<tr>
<td>Lists and tables</td>
<td></td>
</tr>
<tr>
<td>Dimensioning</td>
<td>9, 10</td>
</tr>
<tr>
<td>Sample program</td>
<td>10</td>
</tr>
<tr>
<td>Subscripts</td>
<td>9, 10</td>
</tr>
<tr>
<td>Loops</td>
<td>7-9</td>
</tr>
<tr>
<td>Nested loops</td>
<td>8, 9</td>
</tr>
<tr>
<td>Step size</td>
<td>8</td>
</tr>
</tbody>
</table>
Mathematical functions .......... 6, 28-31
Matrices .......................... 48-54
Addition ........................... 49, 50
CON function ....................... 51
Dimensioning ....................... 51, 52
Examples ........................... 52-54
Identity matrix ..................... 50
Inversion ........................... 51
MAT PRINT statement .............. 49
MAT READ statement ............... 49
Matrix statements, list ......... 48, 49
Multiplication ..................... 49, 50
Scalar multiplication ............. 50
Subtraction ......................... 49, 50
Transposition ...................... 50
ZER function ........................ 51
Multiple variable replacement .... 20

Numbers ............................ 6
Rules for printing .................. 23
Numeric variables .................. 6

QN--GÔ TÔ statement .......... 16, 17
Operations ........................ 5

Parentheses ........................ 5
PRINT statement .................. 15, 16
Printing
   Format control characters ...... 24
   Formatted line output .......... 24-28
   Image statement ................. 24-28
   PRINT statement ................. 21, 22
   PRINT USING statement .......... 24-28
   Rules for printing numbers .... 23
   TAB function .................... 23
Priorities for computing ....... 5, 6
Program
   Definition ........................ 1
   Examples
      Greatest common divisor ... 32, 33
      Illustrating debugging ...... 11-14
      Two equations in two variables . 2-5

Using BCD files ................... 55-57
Using binary files ............... 57, 58

READ statement ................. 3, 4, 14
REM statement ..................... 35, 36
RESTORE statement .............. 36

Spaces in programs .............. 3

Statements
   CALL .......................... 33-35
   CHAIN ........................ 36, 37
   DIM ........................... 10, 18
   END ........................... 18
   FOR and NEXT ................. 17, 18
   FOR and NEXT ................. 17, 18
   GÔ TÔ ........................ 16
   GOSUB and RETURN .......... 32, 33
   IF--THEN ....................... 3, 17
   INPUT ........................ 15
   LET ........................... 3, 14
   QN--GÔ TÔ ..................... 16, 17
   PRINT ........................ 15, 16
   READ and DATA .............. 3, 4, 14
   REM ........................... 35, 36
   RESTORE ....................... 36
   STÔP ........................ 35
   STÔP statement ............... 35

Strings see also Alphanumeric data
   Definition ........................ 19
   String size ..................... 19
   String variable defined ....... 19
   Subscripted variables .......... 9, 10
   Symbols of relation ........... 7

Tables see Lists and tables
   Using the system ............... 66-68
   Control commands ............. 67, 68
   Special keys and controls .... 66, 67

Variables
   Multiple variable replacement .. 20
   Numeric ........................ 6
   String ........................ 19
   Subscripted .................... 9, 10
Computer Centers and offices of the Information Service Department are located in principal cities throughout the United States.

Check your local telephone directory for the address and telephone number of the office nearest you. Or write . . .

General Electric Company
Information Service Department
7735 Old Georgetown Road
Bethesda, Maryland 20014

GENERAL ELECTRIC
INFORMATION SERVICE DEPARTMENT