System 88 Disk BASIC: A Manual

PolyMorphic
Systems

460 Ward Drive Santa Barbara California 93111 (805) 967-2351

This manual describes the BASIC software provided on the System 88 System Disk, PolyMorphic Systems Part Number 820102. The version of BASIC that this manual describes is the second release of PolyMorphic Systems Disk BASIC. The software described herein was created by R. T. Martin and T. A. Crispin. The manual was written by Robin C. Soto, T. A. Crispin, and R. T. Martin, and was edited by Dr. Gerald A. Bradley.

This manual is PolyMorphic Systems Part Number 810140.

A NOTE TO THOSE WHO HAVE BEEN WRITING PROGRAMS WITH THE ORIGINAL POLYMORPHIC SYSTEMS DISK BASIC

This manual describes the second version of PolyMorphic Systems disk BASIC. The new version of BASIC is a greatly improved re-write of our original version, with many new features we believe you'll find useful, as well as improvements to the existing features.

We want to direct your attention to the items in the new BASIC that may affect the programs you wrote using the original version. Each of these items in the new version is either completely new or substantially changed from the original. In some cases, programs written with the old BASIC will not run with the new BASIC unless they are edited. You should find when you read the descriptions of the new features that the differences (and the procedures for editing old programs to run with the new BASIC) are self-evident. To summarize:

1) If you indexed strings using the subscript syntax, you will have to convert those program lines to the new method of indexing strings (using string functions MID$, LEFT$, and RIGHT$). Since the new BASIC uses string arrays, you may want to convert your string indexing program lines to string array statements.

2) You can now dimension your arrays to begin with the first or the zeroeth element. If you want all arrays to begin with index Ø, use the DIMØ statement at the beginning of your program.

3) Note that the Boolean logical operators function a little differently now when applied to data: they treat each piece of data as a 16-bit integer.

4) The new BASIC occupies more memory space than the old version.

5) When you load in a BASIC program, the current contents of memory are NOT erased. In this way you can concatenate BASIC programs. You can now formally merge BASIC programs using CHAIN.

6) You can now make multiple assignments in BASIC.

   EXAMPLE:

   ```
   A,B,C,D,E,F=Ø
   ```

   To save memory space, you may wish to convert your present assignment statements to the above format where appropriate.

7) WAIT and PAUSE have now been added to BASIC. If you have programs that check the keyboard port to see if a character has been inputted by the user of your program before continuing execution of your program, you may want to use WAIT. If you have "time out" loops that count down a certain amount of time before continuing execution of your programs, you may want to use PAUSE.

8) The use of the random number generator function has changed. See Section 6 for more information.

9) You can now include format specifications within the argument of a STR$ function.

10) You can now perform many matrix functions (e.g., MAT PLOT, MAT PRINT, MAT READ, MAT IF, etc.).

11) There are now special array functions: SUM, PROD, MAX, MEAN, STD, and MIN.

12) The Run-Time-Environment is now saved. This means that if you interrupt a program, you can reassign a value to a variable and continue execution of that program. The values of the variables in a program are not cleared unless you use the CLEAR statement or begin execution of a program from its first line.

13) The new version of BASIC includes a file-management system that lets you back up your BASIC programs on tape.

14) If you used the PLOT feature within a FOR-NEXT loop, you may want to convert those program lines to use the MAT PLOT feature. You will find that this change greatly increases the speed of your plot.

15) You will find that many new scientific functions have been added to BASIC. You may want to replace your subroutines that calculate these functions with the appropriate BASIC functions.

16) There are now debugging statements that you can incorporate into your program (e.g., ON ESCAPE, ON ERROR, DUMP, etc.). See Section 10 for more information. You can also single-step a program.

17) A data record (the characters between two carriage returns) can now be any length (not 128 characters or less, as before).

CONFIGURING THE NORTH STAR FLOATING POINT CARD

PolyMorphic Systems provides several versions of BASIC that can be used with the floating point arithmetic processor card available from North Star Computers, Inc. If your version of BASIC is intended for use with the North Star floating point card, it is so labeled.

The floating point card speeds up BASIC arithmetic operations by a factor of from two to thirty, depending upon the complexity of the function involved. A floating point addition (eight-digit precision) will execute twice as fast. The SIN function (eight digits) will execute thirty times faster.

The PolyMorphic BASIC versions intended for use with floating point hardware and other PolyMorphic BASIC versions are completely compatible with each other; programs written with one BASIC can be run with the other with no modifications. After you have installed the North Star card, you can run either kind of BASIC in your computer.

To use the North Star floating point card (FPB-A) with PolyMorphic's BASIC, you must first install several jumpers on the North Star card.

ADDRESS SELECTION

There are six pads labeled 12, 13, 14, G, and H on the card. See the FPB manual section on address selection for their location. Wire the card for addresses 1F00 to 1FFF as follows:

1. Jumper pad 12 to pad G.
2. Jumper pad 13 to pad H.
3. Jumper pad 14 to pad H.
4. Jumper pad 15 to pad H.

PHLDA JUMPER

On the back of the FPB card, connect a jumper from IC3A pin 2 to IC3A pin 7. See the North Star FPB documentation for pin locations.

The floating point card is now ready for installation into your System 8810 or System 8813 and into a POLY 88 that has been modified as described below. See the POLY 88 or System 88 User's Manual for installation instructions.

POLY 88 CPU CARD MODIFICATIONS

The central processor card in the POLY 88 may have to be modified for use with the North Star card. If this modification has not already been performed, remove the CPU card and modify it in accordance with the modification drawing (CPU Modification -SM1 & WAIT) attached. Then re-install the CPU card.

TABLE OF CONTENTS

PART II: BASIC AND THE DISK SYSTEM

Section 1

INTRODUCTION

BASIC has become the language of choice for micro-computer users. It is a relatively easy computer language to learn, yet many sophisticated applications programs are written in BASIC. Its original developers, Dartmouth College Professors Kemeny and Kurtz (1963), conceived of it as a language simple enough to be used by beginners yet powerful enough to carry out complex computation.

The System 88 system disk includes a BASIC interpreter that lets you create and run programs in BASIC. BASIC itself is invoked by typing BASIC and a carriage return. The BASIC files you create are automatically tagged with a .BS filename extension, and when a file with a .BS extension is run, the system (again automatically) brings in BASIC to run it.

The "BASIC file" on your system disk is itself actually a file written in machine language, which can accept instructions and data expressed in the BASIC language and interpret them so that the computer's central processor can understand them.

There is no single official version of BASIC actually in widespread use. A minimal version is recognized by the American National Standards Institute as common to all extended versions, but each extended version is different. This manual describes the Extended BASIC which is a part of the PolyMorphic Systems disk-based microcomputer system and which is provided on the System Disk. If you already know how to program in BASIC, this manual should give you all you need to begin creation of a BASIC program using your PolyMorphic Systems microcomputer. If you are an absolute beginner, you will want to read one of the many books aimed at the novice-- such as "My Computer Likes Me (When I Speak to It in BASIC)"-- before reading this manual.

1.1 MANUAL CONTENT

The information in BASIC: A Manual is grouped into sections, each section building upon information provided in previous sections, so that the novice can develop, section by section, a coherent sense of BASIC and its potential. Anything you do not understand in an early section of this manual is probably clarified by a later section. This manual has also been designed to permit quick, complete referencing by the advanced user. The manual sections are:

PART I: THE BASIC LANGUAGE

Part I incorporates everything that describes the current version of BASIC itself, exclusive of the procedures for working with it on the PolyMorphic Systems microcomputer.

Section 2: Getting Into BASIC. This section deals with the primary elements for building a BASIC program, such as techniques for creating and editing text, and discusses direct statements.

Section 3: Inputting Your Program. Section 3 deals with how you actually type in your BASIC program and provides information on program line numbers and multiple-statement lines.

Section 4: Running Your Program. This section discusses the various control commands you can use when you run your BASIC program.

Section 5: Program Statements. The many types of program statements you can include in your BASIC program are discussed in this section.

Section 6: Functions and Subroutines. This section discusses functions intrinsic to System 88 BASIC, as well as the concept of user-defined functions. Section 6 also deals with the concept of subroutines.

Section 7: Strings and Arrays. This section talks about the concept of strings and arrays and how to use them in BASIC.

Section 8: The MAT feature. This section describes the unique System 88 BASIC matrix feature.

Section 9: The PLOT Feature. The System 88 BASIC PLOT feature is described and demonstrated in this section.

Section 10: Optimizing Your BASIC Program. This section discusses ways you can speed up your BASIC programs and increase their efficiency.

Section 11: Debugging Your BASIC Program. This section discusses the methods you can use to "debug" (fix) your BASIC programs, using the powerful debugging tools provided by PolyMorphic's System 88 BASIC.

PART II: BASIC AND THE DISK SYSTEM

Part II describes the actual use of PolyMorphic Systems Disk BASIC.

Section 12: File Channels. File channels are the paths to and from BASIC and other parts of the system (for example, BASIC data files, printer, video screen) over which the system transfers data.

Section 13: BASIC Data Files: Overview. Data files contain the data or quanitites manipulated by BASIC programs and created as the results of their computations.

Section 14: Creating and Using BASIC Data Files. This section explains how to store and access the data generated and manipulated by a BASIC program.

Section 15: Using the printer from BASIC. You can run a printer directly from a BASIC program by using the technique described here.

Section 16: Sample Programs and Summary of BASIC File-Handling Commands.

Appendix A: Error Messages Generated by BASIC. This appendix lists the error messages generated by BASIC, along with possible causes for those messages.

Appendix B: Loading BASIC and Loading and Saving a BASIC program.

Appendix C: Sample Programs. This appendix contains sample programs which demonstrate the knowing use of the various features of PolyMorphic Systems BASIC.

Appendix D: The BASIC Character Set. The character set for PolyMorphic Systems BASIC is given in this appendix, including ASCII.

Appendix E: Interfacing with the Assembler and Memory. This appendix discusses methods for interfacing BASIC and assembly programs. It also shows you how to directly access memory.

Appendix F: Commands, Statements, Functions, and Keywords Recognized by BASIC.

1.2 THE EXAMPLES IN THIS MANUAL

This manual was written on a PolyMorphic Systems System 88 computer and printed on a Diablo HyType II 1620 printer linked to a System 88. The examples of computer printouts resemble the characters put out by a printer or on the video screen. As you read the manual, sit down with the system and try the examples given in each section; many aspects of BASIC which are not clear in the text will become clear.

In most of the examples, the word "enter" appears opposite the first line of the example. Type in the information located on the line across from "enter" just as it appears in the example.

The part of the example marked "output" indicates the computer's response to the "enter" section. When you have typed the "enter" section of the example correctly, hit the carriage return key at the end of the "enter" section of the example, and the "output" will appear on the video screen. If you make a mistake entering the example, refer to Section 2.1.4.

REM

You will often see the word REM appear in a program line in the examples. This word indicates to the computer that a remark is to follow, not an instruction. BASIC ignores everything on a program line after the word REM. The remark is simply reproduced when the program is displayed. The comments after the REMs appearing in the examples are designed to help clarify the examples for you.

PART I: BASIC

Section 2

GETTING INTO BASIC

The System 88 disk-based microcomputer includes BASIC as part of the system disk, so you do not have to load BASIC as a separate step; just type BASIC while in Exec (i.e. when you see the system prompt $ or $$).

After BASIC is properly loaded into your machine, a message appears on the screen telling you which version of BASIC has been loaded. Also, a prompt symbol > appears at the left side of your monitor screen, telling you that BASIC is ready to receive your input.

In order to use the examples provided with this manual, you must be acquainted with the keyboard and display.

2.1 THE KEYBOARD AND DISPLAY

The computer keyboard works much like a standard typewriter. The shift key on the keyboard functions like a typewriter shift key. Some keyboards have only upper-case letters and use the shift key only for the symbols that appear above the numbers and for other special symbols. System 88 keyboards are full upper and lower-case keyboards, so the shift key affects the full keyboard, letters and other keys alike. Note that on the System 88 keyboard, the CAPS LOCK key affects the letter keys only.

As you strike the keys, the character for each key appears on the video display.

2.1.1 Giving Instructions to BASIC

You can give BASIC some simple instructions in two major ways, by means of a direct statement or by means of a program.

BASIC will execute some instructions immediately; this is the case with direct statements. Some examples of legal, acceptable forms of these instructions are provided in Section 3.

An example of a direct statement:

```
              >
              >
       enter  >PRINT 3+6
       output  9
              >
              >
```

Another way of giving BASIC instructions is to give BASIC a program. A BASIC program consists of a series of statements treated as a unit. BASIC does not execute these instructions immediately and individually. Instead, the instructions in a program are executed sequentially when the program "runs."

To signal BASIC that an instruction is not to be performed immediately, but as a part of a program, the instruction must be preceded by a program line number. Section 3, Inputting Your Program, also provides details regarding construction of a program.

Example:

```
                >
                >
      enter     >10 PRINT 3+6
                >20 PRINT 34-16
                >RUN

      output     9
                 18
                >
                >
```

2.1.2 Carriage Return

To end an instruction to BASIC, type a carriage return (RETURN or RET on most keyboards). This tells BASIC it may go ahead and execute your instruction or (in the case of a program line) store it for later execution. BASIC then returns with a prompt, indicating that it is ready for another instruction.

2.1.3 Interrupting BASIC

To interrupt any process in BASIC, use the Control-Y command: hold down the Control key (CTRL) and type Y. If you were typing a line when you used Control-Y, BASIC will ignore that line and return with a prompt. If BASIC was in the process of executing an instruction, it will finish execution of that instruction and return with a prompt. (Some input/output instructions are interrupted during execution.)

2.1.4 What To Do If You Make A Mistake

If you type in something wrong, BASIC provides several ways of taking it out again. The table below summarizes the deletion commands available in BASIC:

To delete:

Individual characters:	Use the DELETE or RUBOUT key to back-space over the characters you wish to delete. Then retype.
Entire words:	Hold down the Control key (CTRL) and type W. This deletes one word at a time from the current line. Then retype.
Entire lines:	Hold down the Control key (CTRL) and type X. This deletes the entire line that you are typing. A

Control-Y command may also be used. Control-Y will cause BASIC to ignore everything on the current line, although it will not disappear from the screen until the program is relisted. After either of these commands, the correct line may then be retyped.

2.2 PRIMARY ELEMENTS OF A BASIC INSTRUCTION

The primary elements of a BASIC instruction consist of operators and operands. Operators are symbols that cause operations to occur; operands are the entities operated upon. Other elements of BASIC instructions and program lines are discussed in other sections of this manual.

2.2.1 Operators

Operators consist of symbols used to perform certain operations. These operations fall into three broad categories: 1) arithmetic, 2) relational, and 3) logical (or Boolean).

2.2.1.1 Arithmetic Operators

BASIC executes arithmetic operations in response to the following symbols. If several are used in the same expression, BASIC executes them in the order listed:

Example	Symbol	Operation
> >PRINT 9^2 81 > >	^	Exponentiation. On keyboards without this symbol, a Shift-N is used.
>PRINT 7*9 63 > > >	*	Multiplication
>PRINT 6/4 1.5 > >	/	Division
>PRINT 23 + 67 90.97 > > >	+	Addition
>PRINT 567 - 56 511 > >	-	Subtraction

Multiplication and division are equal in precedence; addition and subtraction are also equal in precedence. The order of execution of multiplication and division or of addition and subtraction within the same expression is from left to right. Parentheses may be used to alter the order of execution. When parentheses are used, operations are executed from the innermost parenthesis outward.

Example:

```
>
>REM Show order of expression evaluation and
>REM effect of parentheses.  Note:  Order of
>REM operation execution given in table above.
>PRINT 3+4/7
 3.5714286
>REM Note that division was done first as if
>REM we had said:
>PRINT 3+(4/7)
 3.5714286
>REM So we would need parentheses to get the
>REM expression to be:
>PRINT (3+4)/7
 1
>REM The same thing happens with the expression:
>PRINT 5-3^2
 -4
>REM It was executed as:
>PRINT 5-(3^2)
 -4
>REM The exponentiation (^) was done first, instead of :
>PRINT (5-3)^2
 4
>REM This forces the subtraction to be done first.
>REM Try some examples of your own to see how this works.
```

2.2.1.2 Relational Operators

BASIC evaluates relational operations in response to the following symbols:

Symbol	Operation
=	equals
<	is less than
>	is greater than
<>	is not equal to
>= or =>	is greater than or equal to
<= or =<	is less than or equal to

BASIC will evaluate relational operations and respond with a 1 (if true) or a 0 (if false).

```
                        >
Example:  enter         >PRINT 10>0
          output         1
                        >
```

```
               >
        enter  >PRINT 7>7
        output  0
               >

               >
        enter  >PRINT 144=12^2
        output  1
               >
               >
```

Relational operations may also be used in statements in which the command executed depends upon the result of a test operation.

Example:

```
        >
enter   >X=-1
        >IF X>=0 THEN PRINT X ELSE PRINT "Input positive number"

output  Input positive number
```

2.2.1.3 Logical Operators

BASIC can solve problems in Boolean logic using the following operators: AND, OR, NOT. BASIC treats the operands (see below) of a Boolean operator as a 16 bit integer, and returns the 16 bit Boolean result. (Consult a mathematics text if you are unfamiliar with Boolean logic.)

Examples:

```
enter    >10 A=2 \ B=4
         >20 PRINT A AND B
         >30 PRINT A OR B
         >40 PRINT NOT A
         >RUN

output    0
          6
          65533
         >
```

In evaluating relational or operational expressions, the following priorities are observed in determining the order of execution:

1) NOT

2) all arithmetic operations

3) relational operations

4) AND

5) OR

2.2.2 Operands

The data upon which BASIC performs operations are called operands. These operands are given to BASIC either directly, through on-line input, or indirectly, through program statements. Operands may consist of 1) constants, 2) strings, 3) variables, 4) expressions, or 5) special characters.

NOTE: When BASIC stores a number in memory, it represents it with a maximum of eight digits plus an exponent. BASIC rounds off all numbers larger than eight digits. This means that when BASIC adds the two numbers 5000000 + .009, it will return with the incorrect answer of 5000000. In order to represent numbers larger than 99,999,999, BASIC uses the exponentiational notation (or scientific notation) form, in which a power of ten is used to give the order of magnitude of the number.

Examples:

```
 3.76E+02 means +3.76 X 10^02   (+3.76 X 100), or +376
-3.76E+02 means -3.76 X 10^02   (-3.76 X 100), or +376
 3.76E-02 means +3.65 X 10^-02  (+3.76 X .01), or +.0376
-3.76E-02 means -3.76 X 10^-02  (-3.76 X .01), or -.0376
```

2.2.2.1 Constants

A constant is an unvarying quantity. Since the quantity does not vary, it can be represented by a symbol other than a number, such as K.

2.2.2.2 Strings

A string is a group of text characters (blanks may be included) enclosed by quotation marks. All characters within the quotation marks will be reproduced literally by BASIC without being processed. A string may be represented by a string variable which must take the form of an upper case letter of the alphabet optionally followed by a single digit, followed by a dollar sign symbol. For example: A1$ = "This is a string: A1$ is its name"; "This (1+1*(3+SQRT(16))) is a string too"

2.2.2.3 Variables

A variable is a user-defined name which stands for a constant, an expression, another variable, a string, an array, or a function. All numerical variable names consist of one or two characters: an upper case letter of the alphabet optionally followed by a single digit. A string variable name consists of an upper case letter of the alphabet (optionally followed by a single digit) followed by a dollar sign symbol $. The same name may be used to identify different values as long as the values they identify are of different types. For example, it is possible to have a numeric variable A1, a string named A1$, and functions named FNA1 AND FNA1$. These entities have no relationship to one another.

2.2.2.4 Expressions

An expression is a variable, constant, or function which may stand alone or in combination when separated by the symbols for arithmetic operators.

Example:

```
       >
 enter >REM LEGAL EXPRESSIONS
       >X=A+1
       >Y=COS (3)
       >Z=A*5+ (R+COS (4)/10)
       >S1=105
       >
 enter >REM ILLEGAL EXPRESSIONS
       >L=A4+XX
output Syntax error

 enter >Y2=3COS (X)
output Syntax error

 enter >N=A*5+(COS(3)+2)-3)
output Syntax error
```

2.2.3 Special characters

BASIC recognizes certain special characters and strings that do not fit in any of the above groups. They are:

1) PI This is the constant "pi" which BASIC recognizes as the constant 3.1415926.

2) ERR An error code "variable" that is discussed in Section 11.

3) # A special "variable" used in conjunction with certain array functions. See Section 7.

2.3 DIRECT STATEMENTS

Certain direct statements are acceptable to BASIC for immediate execution. These statements are not a part of a BASIC program but may be included in a program as program statements if desired (see Section 5 -- Program Statements). Usually, direct statements are either PRINT statements or are used in combination with PRINT statements.

Direct statements may be used to: 1) print a text string, 2) evaluate and print an expression, 3) assign a value to a variable, or 4) directly examine the value of a variable during program execution.

A. BASIC will directly print a string given to it in the following form: PRINT string

Example:

```
                >
enter           >PRINT "THIS IS A STRING"
output          THIS IS A STRING
                >
```

B. BASIC may be used to directly evaluate and print expressions, if the statement takes the form PRINT expression.

Example:

```
                >
enter           >PRINT 2*PI
output           6.2831852
                >
```

C. A value may be assigned to a variable, and that value used in a further direct statement. These statements take the form

variable=variable, expression, or string
PRINT variable, expression or string

Example:

```
                >
enter           >P=1+3
                >PRINT P+2

output           6
                >
```

D. A direct statement is often used to directly examine the values of certain variables during program execution to diagnose a programming error. It may take the form PRINT variable, or the form IF test condition, THEN PRINT string or variable.

Example:

```
                >
enter           >10 REM SAMPLE PROGRAM
                >20 Y=7\X=5\Z=X+Y\STOP
                >30 PRINT "Z AFTER "STOP"=",Z+20
                >RUN
output          Stop in line 20

enter           >>IF Z=12 THEN PRINT "Z IS OK"
                  ELSE PRINT "OOPS!"
output          Z IS OK

enter           >>CON
output          Z AFTER "STOP" = 32
                >
```

Section 3

INPUTTING YOUR PROGRAM

Every BASIC program consists of a series of program lines containing program statements. (BASIC will not accept a line of more than 80 characters.) Each program line starts with a program line number, so that BASIC will not try to execute it immediately, but will wait until execution of the entire program is requested by the programmer. Then BASIC will execute the program lines in numerical order. (REM statements need not have line numbers and do not load when they have no line numbers.)

This section deals with the actual typing in of your BASIC program. It contains information about line numbers and program lines. For information on loading an existing BASIC program from a disk, see Appendix B. That appendix will also tell you how to save a BASIC program as a disk file.

3.1 PROGRAM LINE NUMBERS

Every program line begins with a line number, which must be an integer from 0 to 65535 inclusive. Any line of text typed to BASIC which begins with a number is processed by the editor as a program line. BASIC ignores blanks or tabs before the line number, and the blank or nondigit that follows a line number terminates that number. Lines do not have to be typed in sequence -- they will be performed in ascending numerical order when the program is executed. When they are listed, they will be listed in numerical order. An error is generated if the line number is not between 0 and 65535, if the program line is too long, or if memory would overflow if BASIC accepted the new line. Error messages are then generated, and BASIC takes no other action on that line.

The techniques for adding, deleting, and replacing program lines are listed below:

A. Adding a new line to a program: Type in a new program line number, followed by your instructions to BASIC. Remember that lines do not have to be typed in numerical sequence. The new line will be accepted if the line number is a legal one and at least one character follows the line number in the program line.

B. Replacing an existing program line: Type in the program line number of the program line you wish to replace. Then type the program statements you want on that program line. BASIC will replace the original program line with your new program line of the same number.

C. Deleting an existing program line: Type the program line number of the program line you wish to delete. Then hit the carriage return key. If a new program line contains only a program line number, BASIC will delete any preexisting program line beginning with that same program line number.

D. Alternate method of deleting lines: To delete a number of sequential lines in a program, type DEL, followed by the number of the first line to be deleted, a comma, and then the number of the last line to be deleted.

Example:

```
                 >
        enter    >10 X=1
                 >20 Z=2\Y=3
                 >30 PRINT X+Y+Z
                 >40 PRINT X+Y
                 >RUN

        output    6
                  4

        enter    >40
                 >LIST

        output   10 X=1
                 20 Z=2\Y=3
                 30 PRINT X+Y+Z

        enter    >DEL 10,20
                 >LIST
                 30 PRINT X+Y+Z
                 >RUN

        output    0
                 >
                 >
```

3.2 MULTIPLE STATEMENTS PER LINE

Multiple program statements may appear on a single line if they are separated by a back-slash \ (SHIFT-L on some keyboards). A line number must appear only at the beginning of the line. If one program line calls for a jump to another program line, BASIC will be able to return to the proper point in that branching program line, even if that branch statement is on a multiple statement line. ("Branching" takes place when you transfer program execution to another program line. Branches can depend upon a test condition, or they can be unconditional. Go to Section 5 for examples of branching statements.)

Example:

```
                 >
        enter    >
                 >110 X=1\A=X+1\GOSUB 2000\PRINT A
                 >
```

After calling the subroutine at line 2000 in response to the GOSUB statement, BASIC, after finishing the subroutine, will return to the proper point in line 110; that is, to the PRINT A statement.

Section 4

RUNNING YOUR PROGRAM: CONTROL COMMANDS

Now that you have learned how to set up a program, you need to know how to run it. This section discusses the control commands you can use to run your program.

These commands also directly affect the execution of the BASIC program or its representation in memory. The control commands which enable the programmer to save and load the BASIC program differ depending on the method of loading and saving a program; see Appendix B: Loading BASIC and Loading and Saving a Program.

4.1 LIST

Use the LIST command when you want to see a BASIC program listed on the screen. The LIST command may be typed in the following form:

LIST optional line number,optional line number

If you don't give any line numbers, the entire program is displayed. If you provide the first line number, the program is listed from that line number to the end of the program. If both line numbers are supplied, the program is displayed from the first line number given to the second line number, inclusive. Finally, if both optional line numbers are the same, just that one line of the program will be displayed.

Examples:

```
        >
 enter  >10 REM SAMPLE PROGRAM
        >15 X=1
        >20 Y=2
        >25 PRINT X+Y
        >
```

```
        >
        >
 enter  >LIST
output  10 REM SAMPLE PROGRAM
        15 X=1
        20 Y=2
        25 PRINT X+Y
        >
```

```
        >
        >
 enter  >LIST 15,25
output  15 X=1
        20 Y=2
        25 PRINT X+Y
        >
```

```
        >
 enter  >LIST 20
output  20 Y=2
        25 PRINT X+Y
```

```
        >
 enter  >LIST 15,15
output  15 X=1
        >
```

An error message will result if you try to list a program line number greater than the last line of your program.

Example:

```
                >
        enter   >10 REM SAMPLE
                >20 X=1
                >30 Y=2
                >40 PRINT X+Y
                >LIST 50
        output  Line number error

        enter   >LIST 20,50
        output  Line number error
                >
                >
```

It is also possible to LIST to devices other than the video monitor screen (such as a printer or disk file) using the syntax

```
LIST:n
```

where n is the channel number of the file or printer. More on this in Section 15.

4.2 REN (RENUMBER)

After you have made many insertions in a program, the line numbers may become very unevenly spaced. To renumber your program lines and even out the differences between line numbers, type REN followed by the optional beginning value, then the optional increment value. The command takes the form REN optional beginning value, optional increment value. All of the program lines will be renumbered by that command. If the first optional value is not supplied, BASIC will begin the program with line number 10. If the second optional value is not supplied, the program will be renumbered by an increment of 10. Both of the values supplied must be positive integers.

Examples:

```
>
>
>10 REM SAMPLE PROGRAM
>12 INPUT X
>70 PRINT X+1
>
```

```
>
>REN
>LIST
10 REM SAMPLE PROGRAM
20 INPUT X
30 PRINT X+1
>
```

```
>
>REN 50
>LIST
50 REM SAMPLE PROGRAM
60 INPUT X
70 PRINT X+1
```

```
>
>REN 100,100
>LIST
100 REM SAMPLE PROGRAM
200 INPUT X
300 PRINT X+1
```

When you renumber a program, BASIC will automatically renumber the line numbers referenced within a program line.

Example:

```
enter   >10 REM SAMPLE PROGRAM
        >20 INPUT Z
        >30 IF Z>=0 THEN GOTO 50
        >40 PRINT "GIVE A POSITIVE #"\GOTO 20
        >50 PRINT "Z=",Z

enter   >REN 50,50
        >LIST
output  50 REM SAMPLE PROGRAM
        100 INPUT Z
        150 IF Z>=0 THEN GOTO 250
        200 PRINT "GIVE A POSITIVE #"\GOTO 100
        250 PRINT "Z=",Z
```

Caution: If a line number referenced within a program is not a valid line number, it will not be renumbered. However, if you renumber the program, it might become a valid line number-- with unpredictable results.

Example:

```
>10 INPUT Z
>20 IF Z>=0 THEN GOSUB 3000
>30 PRINT "TRY AGAIN WITH POSITIVE #"\GOTO 10
>REN 1000,1000

>LIST
1000 INPUT Z
2000 IF Z>=0 THEN GOSUB 3000
3000 PRINT "TRY AGAIN WITH POSITIVE #"\GOTO 1000
```

4.3 RUN

To begin execution of your program, type RUN followed by a carriage return, and BASIC will begin execution at the first line in your program. If you follow RUN with a line number, BASIC will attempt to begin execution at that line number in the program, and will generate an error message if that line number does not exist.

Example:

```
        >
 enter  >RUN 5000
output  Line number error
        >
```

If no line number is supplied, BASIC will begin program execution at the beginning of the program.

NOTE: If you are just learning BASIC, it is not important that you understand Sections 4.3.1 and 4.3.2 right away. After you have read the entire manual and written a few programs, re-read this section.

When you give BASIC the RUN command, a number of things happen before program execution actually starts, depending on whether or not the RUN command has been supplied with a line number.

4.3.1 When RUN is Given a Line Number

Provided that the line number is not the first line of the program, BASIC will begin execution at that line, with no changes in the status of the variables, etc. This is known as preserving the Run-Time-Environment, and allows for better debugging of programs.

4.3.2 When RUN is Not Given the Optional Line Number

The first thing that is done is to clear the variable and string areas. This means:

1) all numeric variables, the first time they are referenced, will have the value zero (although it is not good programming practice to assume this).

2) no array may be referenced without first dimensioning it with a DIM statement (see Section 7 for a discussion of arrays).

3) the random number generator is reinitialized. This means that unless the random number generator is given a new seed (see Section 6.1 on the RND function for details), the same sequence of random numbers will be generated every time that program is executed.

4) the pointer used to access DATA statements for READ (see Section 5.2.2 on the DATA and READ statements) is set to the beginning of the program. BASIC then checks user-defined functions (see Section 6.2) to see that each function is properly defined, and that each multi-line function has an end. Error messages may be generated if there are errors in any of the user-defined functions.

5) all file channels are forced closed.

Then BASIC begins executing the program at the first line.

4.4 CONTROL-Y

To interrupt the execution of your program, hold down the Control (CTRL) key on the keyboard and type Y. The Control-Y command interrupts any process in BASIC. To continue execution of the program, use the continue command CON.

4.5 CON (continue)

The continue command CON enables the programmer to continue execution of a program after an interruption due to a STOP statement in the program or a Control-Y command used during program execution. Type CON after a double prompt to continue. An attempt to use CON when there are no program lines, when the program has been modified after the interruption, or when CLEAR has been used to clear variable and strings, will result in an error message.

Example:

```
                  >
       enter      >10 REM SAMPLE PROGRAM
                  >20 X=1\INPUT "Y?--",Y\STOP
                  >30 PRINT "Y+1=",X+Y
                  >40 PRINT "Y=",Y
                  >RUN

       output     Y?--589.45
                  Stop in line 20
                  >>CON
                  Y+1= 590.45
                  Y= 589.45
                  >
```

When the CON command is used to continue after a STOP, program execution begins at the statement after the STOP statement. When the CON command is used to continue after an interruption caused by a Control-Y command, program execution is continued after the statement interrupted unless that statement was an INPUT command. In that case, execution resumes at that INPUT command.

Example:

```
                  >
       enter      >10 REM SAMPLE PROGRAM
                  >20 X=1\INPUT "Y?--",Y\PRINT "Y+1=",X+Y
                  >30 PRINT "Y=",Y
                  >RUN

       output     Y?--345.6Y (Control-Y command used here)
                  Interrupted in line 20
                  >>CON
                  Y?--345.67
                  Y+1= 346.67
                  Y= 345.67
                  >
```

Note that in the above examples a double prompt >> appears after an interruption. This indicates that BASIC can continue execution of the program. The double prompt will continue to appear until BASIC can no longer continue execution after modification in the program, use of CLEAR, etc., at which time it will be replaced with a single prompt >.

4.6 CLEAR

Use of the CLEAR command sets all input variables to 0 and all input strings to a null value. It also closes all file channels.

4.7 SCR (SCRATCH)

The command SCR, typed after a prompt, erases all information in working memory: your program and its data. It also closes all file channels.

4.8 DEL (DELETE)

The command DEL is used to delete selected lines from a user program. The correct syntax is:

```
DEL line number, optional line number
```

DEL will start with the first line number and delete all lines up to and including the second line number. If no second line number is given, BASIC will delete only the first line. Note that this differs from the way in which LIST works.

4.9 XREF (CROSS REFERENCE)

The XREF command is a debugging tool that lets you cross-reference the variables in your program with the line numbers in which they appear. It will be more fully explained in Section 11 on the debugging features of BASIC.

4.10 WALK (SINGLE STEP)

The WALK command allows a user to execute his program one line at a time. It will also be discussed in Section 11.

4.11 SUMMARY OF CONTROL COMMANDS

CLEAR	Resets all input variable values to 0 and input strings to null value.
CON	Resumes execution of a program after a STOP or an interruption.
Control-Y	Interrupts any process in BASIC, including program execution.
DEL	Deletes selected program lines.
LIST	Lists program.
REN	Renumbers program lines.

RUN	Begins execution of a program either at the beginning of the program or at the optionally supplied line number.
SCR	Erases the program and anything else typed from the terminal, along with any data calculated by the program.
WALK	Single steps through a program.
XREF	Cross references variables with line numbers.

Section 5

PROGRAM STATEMENTS

Program statements are by far the most important part of BASIC. Program statements make up the instructions which BASIC will follow when it executes a program.

This section of your manual covers the statements in BASIC under several different headings:

1) General program statements.
2) Program statements used to input data.
3) Program statements used to output data.
4) Program statements involved in FOR-NEXT loops.
5) Program statements used to alter program execution.

For sample demonstrations of program statements, see Appendix C: Sample Programs.

5.1 GENERAL PROGRAM STATEMENTS

The three program statements used very commonly throughout any program are discussed below: 1) REM (remark), 2) STOP, and 3) Assignment Statement (LET).

5.1.1 REM (Remark)

The remark statement allows the programmer to add comments to the program without those comments being processed by BASIC. A REM statement may be placed anywhere on a program line, since BASIC ignores everything to the right of it, including the letters "REM." BASIC will, however, print the REM statement when the program is listed. (You may insert REM statements without line numbers into a disk file; without numbers, they are not actually part of the program, and therefore will not load and will not be listed.) The REM statement, unless it is the first statement on the program line, must be preceded by a back-slash \ .

5.1.2 STOP

Insert the STOP statement in a program whenever you want a permanent or recoverable halt. To continue execution from a STOP, use the continue command CON, described in Section 4.5.

5.1.3 Assignment Statement (LET)

Use an assignment statement to set a variable to a given value or expression. The usual form is variable=constant, variable, or expression; for example: A=19. This example sets the variable A equal to 19. The expression on the right of the equals sign can be quite complex; in any case, the expression on the right is evaluated and assigned to the variable on the left.

Example:

```
                >
        enter   >10 A=1320
                >20 B=12
                >30 C= A/B+10.2
                >40 PRINT C
                >RUN

        output   120.2
                >
```

There are two major types of assignment statements: one for numerical variables, as in the examples above, and a second type for string variables.

Example:

```
                >LIST
        enter   10 A$="HOT FUDGE"
                20 PRINT A$
                30 B$=" SUNDAE "
                40 PRINT B$
                50 PRINT A$+BS
                60 PRINT B$+A$
                >RUN

        output  HOT FUDGE
                 SUNDAE
                HOT FUDGE SUNDAE
                 SUNDAE HOT FUDGE
                >
```

The optional keyword LET may be used to indicate an assignment statement. Its use is not encouraged, since it is only a mnemonic device and wastes space on a line. The following examples are identical in meaning.

Example:

```
                >
        enter   >A=X+1
                >LET A=X+1
                >
```

Multiple assignments:

Polymorphic's System 88 BASIC lets you assign a number of variables to the same value at the same time. For instance, you may want to reset some variables to 0. You can do this by putting all the variables to left of the = and separating them with commas, thus:

```
A,B,C,D,E,F=0
```

is equivalent to:

```
A=0         C=0         E=0
B=0         D=0         F=0
```

The order of assignment is right to left. Thus, in the example

```
I=4
J,K,I,B(I)=3
```

B(4) is set equal to 3; then I is set equal to 3, and so forth.

Just as in normal assignment statements, the right hand expression may be arbitrarily complex. Multiple assignments also work with string variables.

```
A$,B$,C$="HELLO!!"
```

If you mix string variables with numeric variables in the list to the left of the =, BASIC will display a syntax error message.

5.2 INPUTTING DATA

The following section discusses the various program statements used to make data available to the program. Data may be made accessible either through direct input from the user terminal (INPUT AND INPUT1) or indirectly from the program itself (DATA, READ, RESTORE).

5.2.1 INPUT and INPUT1

The INPUT and INPUT1 statements are used to ask for data from the user terminal. A question mark is printed by BASIC to prompt the user.

Example:

```
                >
        enter   >10 INPUT X$
                >20 PRINT "The word is: ",X$
                >RUN

        output  ?me
                The word is: me
                >
```

An optional input string may be used as a prompt to the user, in which case BASIC does not print a question mark. If more than one variable is asked for in one input statement, they must be separated by commas, blanks, or tabs.

Example:

```
                >
        enter   >10 INPUT "Give me two numbers--",X,Y
                >20 PRINT "Their sum is: ",X+Y
                >RUN

        output  Give me two numbers--2.5,5.89
                Their sum is: 8.39
                >
                >
```

The INPUT1 statement acts in the same way as an INPUT statement, except that the usual carriage return echo is eliminated. This has the effect of leaving BASIC on the same line as the input, so that the next input prompt or message printed by a PRINT statement will appear on the same line as the first INPUT1 statement.

Example:

```
                >
        enter   >LIST
                10 INPUT "Your name? ",N$
                20 INPUT1 "Give two numbers--",S,S1
                30 PRINT " Hi, ",N$
                40 PRINT " The sum is : ",S+S1
                >RUN

        output  Your name? Robin
                Give two numbers--345.78,896.51 Hi, Robin
                 The sum is:  1242.29
```

5.2.2 Inputting From Disk Files

It is possible to input data into BASIC programs from a disk file. This capability is discussed in Section 13.

5.2.3 DATA and READ

The DATA and READ statements are used to ask for data from within the program itself. The DATA statement contains within it the actual data that the program uses during execution. The DATA statement may contain either string or numerical data. The data must be separated by commas, and strings must be enclosed by quotation marks. The data in the DATA statement are read by the READ statement and must be consistent with the type of variables (numerical or string) used in the READ statement, or an error message will be generated.

When the first READ statement in a program is encountered, a pointer is set to the first piece of data in the first DATA statement in the program. Every time a READ variable reads one piece of data, the pointer advances to the next piece of data. As all data from the first DATA are read, the pointer advances to the first piece of data in the next DATA statement, and so on, until all READ variables have been matched with data. If there are more data than needed, the remaining unread data are ignored. If, however, there are fewer data than there are READ variables (that is, the pointer is out of data), an error message will be generated.

Examples:

```
                >
        enter   >100 READ A,B,C\PRINT "A,B,C: ",A,B,C
                >200 READ X,Y,Z\PRINT "X,Y,Z: ",X,Y,Z
                >300 DATA 1,2,3,100
                >400 DATA 200,300
                >RUN
```

```
output  A,B,C:  1 2 3
        X,Y,Z:  100 200 300
        >

        >
enter   >10 READ A$,B$,C$\PRINT A$,B$,C$
        >20 PRINT C$,A$,B$
        >30 DATA " WE "," ARE "," HERE "
        >RUN

output  WE ARE HERE
        HERE WE ARE
        >
        >
```

5.2.4 RESTORE

A RESTORE statement allows the programmer to change the order in which READ statements access DATA statements. Use of the RESTORE statement enables the programmer to direct a particular READ statement to a particular DATA statement. The RESTORE statement takes the form RESTORE optional line number. If you omit the optional line number, the READ statements begin reading data from the first DATA statements in the program. With the line number included, the READ statements are directed to a DATA statement on that or a following line.

Example:

```
        >
enter   >10 READ A,B,C\PRINT "A,B,C: ",A,B,C
        >20 RESTORE
        >30 READ X,Y,Z\PRINT "X,Y,Z: ",X,Y,Z
        >40 DATA 1,2,3
        >50 DATA 100,200,300
        >60 DATA 5,6,7
        >RUN

output  A,B,C:  1 2 3
        X,Y,Z:  1 2 3
        >

enter   >10 RESTORE 50
        >20 READ A,B,C\PRINT "A,B,C: ",A,B,C
        >30 READ X,Y,Z\PRINT "X,Y,Z: ",X,Y,Z
        >40 DATA 1,2,3
        >50 REM READ DIRECTED TO THIS LINE
        >60 DATA 100,200,300
        >70 DATA 5,6,7
        >RUN

output  A,B,C: 100 200 300
        X,Y,Z: 5 6 7
        >
```

5.2.5 Single Character Input Functions INP(0) and INP(1)

The functions INP(0) and INP(1) allow the user to test for characters in the input buffer and input single characters from the keyboard. The function INP(0) returns 0 if there are no characters waiting in the input buffer to be read. INP(1) returns the integer value of the next character from the keyboard buffer, without echoing it to the screen. (See Appendix D for the values asigned by the ASCII code to the full set of characters.)

Example:

```
enter   100 REM DEMONSTRATE INP(0) TESTING FOR INPUT
        110 PRINT "You have 10 seconds to type cow"
        120 PRINT "?",
        130 Z=TIME(0) \ REM RESET CLOCK
        140 IF INP(0)>0 THEN 190 \ REM SOMETHING TYPED
        150 IF TIME(1)<10*60 THEN 140
        160 REM TOO LONG. COMPLAIN
        170 PRINT "...Too late, you didn't type cow"
        180 GOTO 110
        190 INPUT "",A$\ IF A$="COW" THEN 210
        200 PRINT "You didn't type cow"\ GOTO 110
        210 PRINT "Thank you."
        >RUN

output  You have 10 seconds to type cow
        ?...Too late, you didn't type cow
        You have 10 seconds to type cow
        ?frog
        You didn't type cow
        You have 10 seconds to type cow
        ?cow
        Thank you.
        >
```

5.3 OUTPUTTING DATA

There are several ways you can change the format of data output by a program. All of these involve the use of PRINT statements. This section will briefly outline the use of the free-format PRINT statement, the use of the TAB function in formatting data, and the use of format strings to set up data formats.

5.3.1 PRINT

The PRINT statement prints out the one or more elements in its print list. The elements must be separated by commas. If there are no elements in a print list, that is, if the word PRINT is alone on a line, BASIC will print an empty line. PRINT statements will evaluate and print expressions (including intrinsic functions) and variables. A string in the print list is printed as given, but without the surrounding quotation marks.

Example:

```
        >
enter   >10 PRINT "RUBBER CHICKEN", SQRT (100),2+2
        >15 PRINT "SECOND LINE"
        >RUN

output  RUBBER CHICKEN 10 4
        SECOND LINE
        >
```

If the last element in the print list is followed by a comma, a carriage return is not printed, and the output of the next PRINT statement or INPUT statement will appear on the same line as the original PRINT statement output. If the output of a PRINT statement is too long to fit on the current monitor output line, it will be continued on the next line with no carriage return being generated. The PRINT statement may take the form PRINT print list. The print list may contain strings, variables, or expressions, all separated by commas, with strings being surrounded by quotation marks.

5.3.2 Formatting the PRINT Statement

If you do not specify any formatting in a PRINT statement, the data is printed in the default free-format style. In the free format, all data in the print list are printed left-justified with the prompt symbol, and all numerical elements are printed and separated by a blank. Unless a specific format is given by the programmer, BASIC prints all numerical data in the default format given below.

The Default Format

(For a discussion of exponential form or scientific notation, see note following Section 2.2.5.)

1. Numbers eight digits long or less and in non-exponential form will be printed as given.

   Example:

   ```
           >
   enter   >PRINT 12.34567
   output   12.34567
           >
   ```

2. Numbers longer than eight digits and in non-exponential form will be rounded off to eight significant digits and printed in standard exponential form.

   Example:

   ```
           >
   enter   >PRINT .00123456789
   output   1.2345679E-03
   ```

3. Numbers in exponential form eight digits long or less will be printed in non-exponential form if doing so would result in a number of eight digits or less.

Otherwise, the number is printed in standard exponential form.

Example:

```
        >
enter   >PRINT 123.45E+05
output   12345000
        >
enter   >PRINT 123.45E+06
output   1.2345E+08
        >
        >
enter   >PRINT 123.456E-05
output   .00123456
        >
```

4. Numbers in exponential form longer than eight digits are rounded off and printed in non-exponential form if doing so would result in a number of eight digits or less. Otherwise the number is printed in standard exponential form.

Example:

```
        >
enter   >PRINT 123.4567891E+06
output   1.2345679E+08
        >
        >
enter   >PRINT 123.4567891E+05
output   12345679
        >
```

TAB

The TAB function provides a way to space output across the screen. The TAB statement takes the form PRINT TAB (expression), print list. TAB evaluates the expression within its parentheses and moves over to that character position on the screen (counting from the left; first position is 0) before printing the elements in the print list. The TAB value can be greater than 64, but must be less than 256 and positive.

Example:

```
        >
enter   >10 PRINT TAB(15),"UNIT ONE",TAB(25),"UNIT TWO",
        >30 PRINT TAB(19),"A",TAB(29),"B",TAB(39),"C"
        >RUN

output                 UNIT ONE  UNIT TWO  UNIT THREE
                           A         B         C
        >
        >
```

Format Strings

Format strings specify the way numerical data is outputted by a print statement. A format string may appear anywhere in a PRINT statement after the PRINT command, and must begin with a percent symbol (%). An empty format string allows data to be printed in free format. The form of a PRINT statement with a format string is PRINT optional unformatted print list, % optional format characters optional format specification, print list to be printed in specified format. More than one format string may appear in a PRINT statement. An example of a PRINT statement containing the format string C$3I is the following:

```
PRINT "ME," %C$3I, "345"
```

%C$15F2, X

A. Format Characters

- C Places commas to the left of the decimal point as needed.
- $ Places dollar sign symbol to the left of the value printed.
- Z Eliminates trailing zeros.
- # Sets the format string of which it is an element to the new default format for printing numerical data.

Example:

```
        >
enter   >PRINT %C$Z,45678987.590000
output   $45,678,988
```

The format character # sets a new default format. This means that if the format string %C$# is encountered in a PRINT statement, all unformatted numbers in the program after that statement will be printed in that format. To restore the default format to the original, free-format style, the null format string %# is used, either with or without a print list. After the null format string is encountered in a program, the default format reverts to free format.

%C#9F2

Example:

before 9F2

```
enter    10 PRINT\PRINT"In new default format--"
         20 PRINT %$C#,9999
         30 FOR I=2000 TO 5000 STEP 1000
         40 PRINT TAB(30),I,
         50 NEXT
         60 PRINT\PRINT"Reset to old default format--"
         70 PRINT %#,9999
         80 FOR I=2000 TO 5000 STEP 1000
         90 PRINT TAB(30),I,
        100 NEXT
         >RUN

output   In new default format--
          $9,999
                                     $2,000 $3,000 $4,000 $5,000
         Reset to old default format--
         9999
                                     2000 3000 4000 5000
```

B. Format Specifications (for numerical data only)

The format specifications (similar to those in FORTRAN) specify the format in which numbers will be printed on the screen. In the specifications below:

n = number of spaces in the field in which the data are to be printed. The left margin of the field is even with the prompt symbol. n must be less than or equal to 25.

m = number of digits to be placed to the right of the decimal point. (However, if m >8, all digits past the eighth will be zeros.)

1. F-Format: The F-format prints numbers right-justified in a field n characters wide, with m digits to the right of the decimal point. This specification takes the form nFm.

Example:

```
               >
     enter     >PRINT %15F5,3798.6788992
     output         3798.67890
               >
```

2. I-Format: The I-format specification prints only integers (if a non-integer is entered, an error message will be generated). The numbers are printed right-justified in a field n characters wide. This specification takes the form nI.

Example:

```
               >
     enter     >PRINT %10I,2345
     output          2345
               >
```

3. E-Format: The E-format specification prints numbers right-justified in an n-character wide field in scientific notation with m digits to the right of the decimal point.

Example:

```
               >
     enter     >PRINT %10E3,3798.678892
     output     3.799E+03
                >
```

Note: The number 3.799E+03 represents 3.799 X 10^3. (For further discussion of scientific notation or exponential form, see the note in Section 2.2.5.)

Example:

```
               >
     enter     >PRINT 3.799E+03
     output     3779
               >
```

In order to avoid format specification errors, it is important to remember to reserve enough space in the print field by using a large enough n so that the number given to the format specification can be printed. For instance, in the example below, 11 spaces must be reserved in the print field if m = 5 (significant digit, decimal point, m, and the four characters E,+,0,2 = 11 spaces); otherwise an error message is generated.

Example:

```
        >
enter   >PRINT %10E5,234.56
output  Format error
enter   >PRINT %11E5,234.56
output  2.34560E+02
```

5.3.3 Outputting to the Disks and the Printer

These capabilities are discussed in Sections 14 and 15.

5.4 ITERATION: THE FOR-NEXT LOOP

Often in writing a computer program to solve some problem, we find that we would like to perform a certain set of statements a number of times for a certain set of arguments.

Let's say that we wanted to print the integers from 1 to 10 inclusive and their squares. We would write a BASIC program that would execute this process, like this:

Example:

```
        >
enter   >100 REM This program is a loop
        >110 J=1
        >120 IF J>10 THEN GOTO 160
        >130 PRINT "The square of ",J," = ",J^2
        >140 J=J+1
        >150 GOTO 120
        160 PRINT "End!"
        >RUN
```

When we run this program, the variable J is set to 1 by line 110. We then see if J is greater than 10. The first time through, J has the value of 1, so we continue execution with line 130, where we print the value of J, and J squared(J^2). Then we add 1 to the current value of J, and go back to the IF statement on line 120. We "loop" through lines 120, 130, 140, and 150 until J is incremented by line 140 to the value 11. Then, when we perform the IF statement on line 120, J is greater than 10, so we go to line 160, thus terminating the loop.

This "loop" can be thought of as the combination of a number of elements:

1) the "loop variable" J, in the example above, which takes on the values 1 through 10 in the loop.

2) The starting value for the loop variable. In the example, the starting value for J is 1, as set on line 110.

3) A terminating condition; in the example, the loop will terminate, or stop, when J is greater than 10, as detected by the IF statement in line 120.

4) An increment (or decrement) to apply to the loop variable: In the example on line 140, we add 1 to the value of J each time through the "loop", so that during the process of the computation, J takes on the values 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

5) A set of statements that is executed repeatedly, also called the loop body. In the example, the loop body consists of the single PRINT statement on line 130.

6) An indicator marking the end of the loop. In the example, the GOTO 120 statement on line 150 denotes the end of the loop. When the variable J exceeds the terminating condition, 10, as specified by the IF test on line 120, program execution will resume past the end of the loop, at line 160. We could write out this set of statements each time we wanted to execute a statement or set of statements repeatedly, but this would be time-consuming and give us more chances to make programming mistakes. However, this process of "looping" or iteration is done so often that BASIC has a shorthand way of specifying this procedure, with more flexibility, using two statements: FOR and NEXT.

A program equivalent to the one given at the start of this section but using FOR and NEXT looks like this:

```
>
>100 REM FOR-NEXT LOOP
>110 FOR J=1 TO 10 STEP 1
>120 PRINT "The square of ",J," = ",J^2
>130 NEXT J
>RUN
```

We'll go through this new program and identify the same six elements we did in the previous program:

1) The "loop variable." In this case, the loop variable is still J, which appears just after the word FOR on line 110. In general, the loop variable immediately follows the word FOR in a FOR statement, and cannot be a string variable (such as J$; that would be illegal), or have a subscript (such as D(3); that too would be illegal).

2) The starting value. Above, in the FOR statement, we see "J=1," which gives the starting value for the loop, 1, just as in line 110 of the previous program. This starting value can be any expression, and is evaluated only once, at the beginning of the loop.

3) The terminating condition. We see in the program above, using FOR and NEXT, on line 11Ø, the characters "TO 1Ø." This gives the terminating value to test the loop variable (J in this case) as 1Ø, just as it did in the IF statement on line 12Ø of the other program. The terminating value, in this case the number 1Ø, can be any arbitrary numeric expression. It is important to remember, however, that this expression is only evaluated ONCE, at the start of the loop, and not every time through.

4) An increment (or decrement) to apply to the loop variable. In the other program, this was specified in line 14Ø, where we said J=J+1, incrementing J by 1 each time. In the FOR statement the increment is specified by the part of the line that says "STEP 1," defining the increment to be 1. This number also may be any numeric expression, and is only evaluated once, at the start of the loop.

5) A set of statements to be executed repeatedly. In the example using FOR and NEXT, the "loop body" is the single statement on line 12Ø, the PRINT statement.

6) An indicator marking the end of the loop. In the first example, the "loop body" was the single PRINT statement on line 13Ø. In the case of the FOR NEXT loop, the FOR and NEXT statements clearly show what statement or statements will be repeated; that is, any statements that come between the FOR and the NEXT.

The FOR-NEXT statements, then, define the same process and set of elements that we identified in the first case. Yet they provide a quicker, more concise way of specifying a sequence of statements to be repeatedly executed. The FOR-NEXT loop also allows more flexibility, and "hides" the "housekeeping" functions required by the loop we had to specify in the initial program which used the IF statement. Some of the things the FOR-NEXT loop allows us to do are:

1) If we do not give an expression "STEP <exp>" where <exp> is an arbitrary numeric expression, a default step of 1 will be used.

2) The values for the initial value, terminating value, and step size do not have to be an integer or positive. For example, the statement

```
100 FOR W=-1 TO -20 STEP -1
```

would perform some set of statements 2Ø times, with the variable W taking the values -1, -2, -3, -4... to -2Ø.

3) The statements in the loop body may be performed zero times, once, or indefinitely, depending on the conditions and step size chosen.

4) We do not have to specify the variable name on the NEXT statement, although this is quite helpful for debugging (in fact, specifying the variable name slows things down!).

5.4.1 Nesting of FOR-NEXT Loops

Often we like to have an iterative (looping) process going on inside of another iterative process. It is perfectly valid to have one FOR-NEXT loop inside another-- with the following restriction: the inside loop must be totally contained within the outer loop.

Example:

```
               >
    enter      >LIST
               10 REM NESTED LOOPS
               20 FOR J=1 TO 10
               30 FOR K=1 TO 10
               40 PRINT K+(J-1)*10,",",
               50 NEXT K
               60 PRINT
               70 NEXT J
               >RUN

    output     1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
               21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
               31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
               41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
               51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
               61, 62, 63, 64, 65, 66, 67, 68, 69, 70
               71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
               81, 82, 83, 84, 85, 86, 87, 88, 89, 90
               91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
               >
```

This program prints a list of numbers from 1 to 100. The inner loop, as shown above, consists of lines 30, 40, and 50, while the outer loop consists of lines 20 and 70. The number of nested loops is restricted only by the amount of available memory. To see how many FOR-NEXT loops you may nest on your machine, refer to the Sample Program NEST in Appendix B.

The following examples show some of the possibilities with FOR-NEXT loops. Some of these examples show correct usages; others show errors and what BASIC's response will be.

Examples:

```
               >
    enter      >100 REM Normal loop
               >110 FOR I=1 TO 10 STEP 1
               >120 PRINT I,",",
               >130 NEXT I
               >RUN

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

```
        >
enter   100 REM We don't need to specify step
        105 REM or next variable.
        110 FOR W=1 TO 10\PRINT W,",",
        115 NEXT
        >RUN

output   1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
        >

        >
enter   >100 REM Initial value, step, final non-integral
        >110 FOR E=.2 TO 1.2 STEP .3
        >120 PRINT E,
        >130 NEXT E
        >120 PRINT E,",",
        >RUN

output   .2, .5, .8, 1.1,
        >

        >
enter   >100 REM Using negative step value
        >120 FOR E=10 TO 1 STEP -1
        >130 PRINT E,",",
        >140 NEXT
        >RUN

output   10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
        >

        >
enter   >10 REM Negative numbers
        >15 FOR W=-1 TO =11 STEP -1
        >20 PRINT W,",",
        >25 NEXT
        >RUN

output   -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11,
        >

        >
enter   >100 REM FOR NEXT loop all on one line
        >110 FOR I=1 TO 10 \ PRINT I,",", \ NEXT
        >RUN

output   1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
        >

        >
enter   >100 REM Error-no NEXT statement
        >110 FOR I=1 TO 100
        >RUN

output  110 FOR I=1 TO 100
                          ^

        FOR-NEXT error
```

```
enter    >100 REM Error-wrong variable on NEXT
         >110 FOR J=1 TO 100
         >120 NEXT Q
         >RUN

output   120 NEXT Q
                  ^
         FOR-NEXT error
         >

         >
enter    >100 REM Error-string variables
         >110 FOR I$="ONE" TO "THREE"
         >120 NEXT
         >RUN

output   110 FOR I$="ONE" TO "THREE"
                   ^
         Type error
         >
```

5.5 BRANCHING STATEMENTS

It is often desirable to alter the usual order of program line execution. Branching statements are those statements which enable BASIC to jump to program lines out of numerical sequence. This jump may be based on the result of a test condition (conditional branching) or simply be a direct branch (unconditional branching). Most of these statements are frequently used in combination with one another.

5.5.1 GOTO

The GOTO statement lets you transfer execution to another program line. The GOTO statement takes the form GOTO line number.

Example:

```
enter    >
         >10 REM Prints square root of X
         >20 INPUT1 "A number?--",X
         >30 PRINT " Square root of ",X," is: ",SQRT (X)
         >40 GOTO 10
         >RUN

output   A number?--34 Square root of 34 is: 5.8309519
         A number?--56 Square root of 56 is: 7.4833148
         A number?-- (Control-Y command used here)
         Interrupted in line 20
         >>
```

Note that the program above is an infinite loop, and must be interrupted by the user.

5.5.2 ON...GOTO

The ON...GOTO statement allows multiple branching from one program line to many others, depending upon the value of the variable specified. The ON...GOTO statement takes the form ON variable or expression GOTO program line number(s). If the expression or variable after ON evaluates to a 1, BASIC will jump to the first line number listed after the GOTO. If the expression evaluates to a 2, the second line number listed will be taken, and so on. Expressions are truncated to an integer; 1.1 evaluates to a 1.

Example:

```
               >
     enter     >10 FOR X=1 TO 3
               >20 ON X GOTO 30, 50, 70
               >30 PRINT "X equals one"
               >40 GOTO 80
               >50 PRINT "X equals two"
               >60 GOTO 80
               >70 PRINT "X equals three"
               >80 NEXT
               >RUN

               X equals one
               X equals two
               X equals three
               >
```

Note that in the following example, when X is negative a jump is made into program line number 20, when X equals 0 a jump is made to line 40, and when X is positive a jump is made to line 60.

Example:

```
               >
     enter     >10 INPUT X\ON SGN(X)+2 GOTO 20,40,60
               >20 PRINT "Line 20: X is negative"
               >30 GOTO 70
               >40 PRINT "Line 40: X is zero"
               >50 GOTO 70
               >60 PRINT "Line 60: X is positive"
               >70 STOP
               >RUN

               ?-56
               Line 20: X is negative
               >RUN

               ?0
               Line 40: X is zero
               >RUN

               ?456
               Line 60: X is positive
               >
```

(See Section 6, Functions and Subroutines, for an explanation of the SGN function.)

If the expression after ON is less than 1 or greater than the number of program line numbers listed after the GOTO, BASIC will generate an error message.

Example:

```
                >
      enter     >LIST
                10 FOR X=1 TO 4
                20 ON X GOTO 30,40,50
                30 PRINT "You're close"\GOTO 60
                40 PRINT "You're warmer"\GOTO 60
                50 PRINT "You're hot!"
                60 NEXT
                >RUN

                You're close
                You're warmer
                You're hot!

                20 ON X GOTO 30,40,50
                                   ^
                Out of bounds error
```

5.5.3 ON...GOSUB

The ON...GOSUB statement works just like the ON...GOTO statement, except that instead of branching to the indicated line, it executes the subroutine (see Section 6 for a discussion of subroutines) at that line number. After the subroutine has executed, execution continues with the statement following the ON...GOSUB statement.

5.5.4 IF-THEN

The IF-THEN statement is used to set up a test condition which must be met before further instrucions within the IF-THEN statement can be executed. The IF-THEN statement takes the form IF test condition THEN legal IF-THEN clause. The test condition may compare variable to variable, variable to expression, stringto string, etc. Legal IF-THEN clauses include: 1) GOSUB subroutine line number, 2) RETURN, 3) GOTO line number, 4) PRINT print list, 5) ON variable or expression GOTO line number, 6) STOP, or 7) variable= variable, expression, or string.

Example:

```
                >
      enter     >10 INPUT "Want to play? ",A$
                >20 IF A$="no" THEN GOTO 50
                >30 REM Assumes all input other than "no" is "yes"
                >40 PRINT "Here are instructions..."\GOTO 60
                >50 PRINT " O.K. Catch you later"
                >60 REM End of program
                >RUN
```

```
        output  Want to play?  yes
                Here are instructions...
                >RUN

                Want to play?  no
                O.K. Catch you later
                >
                >
                >
```

The IF-THEN statement may perform multiple commands as a result of the test condition. The multiple commands must be written on the IF-THEN statement program line, and separated by back-slashes \ .

Example:

```
                >
                >SCR
                >10 INPUT "Give me a number--",X
                >20 IF X=1 THEN PRINT "Right answer"
                >25 PRINT "Go on!"\GOTO 200
                >30 PRINT "X not equal to one"
                >200 PRINT "This is the end!"
                >RUN

                Give me a number--3
                X not equal to one
                This is the end!
                >RUN

                Give me a number--1
                Right answer
                Go on!
                This is the end!
                >
```

5.5.5 ELSE

An IF-THEN statement may also optionally include an ELSE statement. The ELSE statement includes a legal IF-THEN clause, and may also include another IF-THEN statement. If either the THEN clause or the ELSE clause is a simple GOTO, then the word GOTO may be omitted.

Example:

```
        >
enter   >10 IF X>3 THEN PRINT "X>3" ELSE GOTO 200
enter   >10 IF X>3 THEN PRINT "X>3" ELSE 200
        >
enter   >IF 1=1 THEN PRINT "ONE" ELSE PRINT "OOPS!"
output  ONE
        >
        >
enter   >10 A$="YES"\X=0
        >20 IF A$="YES"THEN IF X=0 THEN PRINT"GO!"ELSE PRINT"WRONG"
        >RUN

        GO!
```

5.5.6 EXIT

The EXIT statement is identical to a GOTO except that it should be used when branching out of a FOR-NEXT loop. This is because it terminates the active FOR loop and reclaims the associated internal stack memory. If an EXIT is not used when branching out of a FOR-NEXT loop, the internal stack could become full and result in a control stack error message.

Example:

enter
```
>10 X=3
>20 FOR I=1 TO 1000
>30 FOR J=1 TO 1000
>40 PRINT I,J
>50 IF X=3 THEN EXIT 200
>60 NEXT\NEXT
>200 PRINT "END"
>RUN

 1 1
END
```

5.6 STATEMENTS MODIFYING PROGRAM EXECUTION

5.6.1 CHAIN

The CHAIN statement in a BASIC program allows users to chain or link BASIC programs one after another, automatically. The Run-Time-Environment is preserved during the chaining procedure. One can think of the chaining procedure as a super-GOTO statement, which branches to another BASIC program not in memory.

5.6.2 DUMP

This command dumps the defined variables (outputs them to the video screen). See Section 11 on debugging for a detailed description.

5.6.3 WAIT

This commands halts program execution, prints the message:

Waiting...

on the monitor, and waits until ANY key is struck before continuing program execution.

5.6.4 PAUSE n

This command will halt program execution for n clock cycles (one cycle is 1/60 sec.) before continuing execution. Thus PAUSE 60 will cause the program to pause in its execution for one second. n may be any expression that evaluates to a number between 0 and 65535. You can end a pause (and return to BASIC) by hitting Control-Y.

5.6.5 ON ERROR

This command provides user control over program errors. A detailed description is in the debugging section.

5.6.6 ON ESCAPE

This command provides user control over panic stops (Control-Y). See the debugging section for a complete discussion.

5.7 SUMMARY OF PROGRAM STATEMENTS

CHAIN	Chains or links BASIC programs.
DATA	Contains data for program execution accessed by READ. Data must be separated by commas and may be either numerical or string in type. Strings must be enclosed in quotation marks.
DUMP	Dumps defined variables to video screen.
ELSE	Used in conjunction with IF-THEN statement. IF test condition THEN legal IF-THEN clause ELSE legal IF-THEN clause or additional IF-THEN statement.
EXIT	Similar to GOTO statement, but should be used when branching out of a FOR-NEXT loop to avoid stack-full error.
FOR-NEXT	Sets up loop within program. Loop is repeatedly executed until specified terminal value is passed by variable given in FOR statement. Unless specified, variable is incremented by +1. FOR loop variable=initial value TO terminal value STEP optional step value.
GOTO	Unconditional branching statement, transferring program execution to specified line number. GOTO line number.
IF-THEN	IF test condition THEN legal IF-THEN clause or additional IF-THEN statement. Execution of statement after THEN depends upon fulfillment of test condition.
INPUT	Inputs data from user of program. May include optional input string as a prompt. Otherwise, INPUT prompts program user with a question mark. INPUT optional prompt string, string or numerical variable.

INPUT1	Identical to INPUT except that carriage return echo (after user input) is eliminated, so that the next PRINT or INPUT statement appears on the same line as original input.
LET	Optional assignment statement. LET variable= variable, expression, or string.
ON ERROR	User-defined error control.
ON ESCAPE	User-defined control of control-Y.
ON...GOSUB	Conditional selection of subroutines. Analagous to ON...GOTO.
ON...GOTO	A conditional statement allowing a branch to a specified line number if a test condition is met. If the variable or expression equals 1, a branch to the first line number listed is taken; if the variable or expression equals 2, a branch to the second line number listed is taken, and so on. ON variable or expression GOTO line number.
PAUSE n	Pause in program execution for n clock cycles.
PRINT	Prints data specified in the print list. The print list may contain elements which are variables, strings, or expressions, all separated by commas. PRINT will evaluate and print expressions and variables, and print literally (not evaluate) strings. A format string (Section 5.3.2) or a TAB (Section 5.3.2) may be included with a PRINT statement to format output. PRINT optional format string or TAB (expression), print list.
READ	Used in combination with a DATA statement to access the data contained in a DATA statement. READ variable list.
REM	Used to place comments within the program. Must be the last statement on a program line, preceded by a back-slash unless it is the only statement on the line. REM comment.
RESTORE	Used to change the order in which a READ statement accesses data from a DATA statement. May optionally include a line number of a particular DATA statement. Otherwise, the READ statement following RESTORE is directed to begin reading data from the first DATA statement in the program.
STOP	BASIC halts execution of a program when it reaches a STOP statement.
WAIT	Wait until the keyboard is struck before continuing program execution.

Section 6

FUNCTIONS AND SUBROUTINES

It is often desirable to perform one section of a program more than once during the execution of a program. Rather than type this section over and over at various points throughout the program, BASIC has some rather ingenious ways of repeating program sections: functions and subroutines.

6.1 INTRINSIC FUNCTIONS

Some commonly used functions have been incorporated into BASIC as intrinsic functions. One of these functions may replace many lines of program statements. The intrinsic function may be used as part of an expression (for example, Z=COS(SQRT(X)*75/100)) or may stand alone (for example, PRINT SIN (X)). The intrinsic functions of BASIC are listed below.

6.1.1 Regular Intrinsic Functions

SQRT(expression)	Returns the positive square root of a positive expression. An expression less than 0 will result in an error message.
EXP(expression)	Returns the value of e (2.71828...) raised to the specified power.
LOG(expression)	Returns the natural logarithm (base e) of the expression.
LOGT(expression)	Returns the logarithm to the base 10 of the expression.
COS(expression)	Returns the cosine of the expression in radians.
SIN(expression)	Returns the sine of the expression in radians.
TAN(expression)	Returns the tangent of the expression in radians.
ABS(expression)	Returns the absolute value of the expression.
INT(expression)	Returns the nearest integer which is less than the expression.
SGN(expression)	Returns 1, 0, or -1 if the sign of the expression is +, 0, or -.
RND(expression)	Returns a random number greater than 0 and less than 1. BASIC generates a sequence of numbers that are randomly distributed, based on a given "seed" value. Where one enters this sequence when using

the RND function depends upon the expression (seed value) given to the RND function. The seed value must be greater than or equal to 0 but less than 1. If the seed value is 0, a point in the sequence of random numbers is chosen depending upon the last random number produced, and a random number is produced. The next time that RND(0) is called within the same program, the next number in the sequence is produced, and so on. If the seed values are the same the next time the program is run, an identical sequence of random numbers will be produced. This is important if the programmer wishes to repeat exactly a simulation of a random process. A non-zero seed value will always produce the same random number. For example, RND(.1) always gives .1640625.

To completely randomize the RND function for every use of the program, the following statement is suggested: RND(TIME(1) /65536). This provides seed values based upon the current value of the real time clock.

The RND function also accepts arguments greater than 1. In this case, it returns a random integer between 1 and INT(N) inclusive.

Example:

```
>SCR
>10 FOR I=1 TO 100
>20 PRINT RND(10),
>30 NEXT
>RUN
```

Compare this with the normal values returned:

```
>SCR
>0 PRINT %#8F5
>10 FOR I=1 TO 100
>20 PRINT RND(0),
>30 NEXT
>RUN
```

TIME(expression)

The TIME function returns as its value the 16 bits of the System 88 real time clock, which is incremented every 1/60th of a second. The expression in the TIME function must evaluate to a value greater than or equal to 0 and less than 65536. If the expression does not evaluate to 0, the current value of the real time clock is returned. If the expression is 0, the

TIME function returns the current value of the real time clock and sets the timer to 0; this is useful for recording elapsed times. Since only 16 bits of the timer are returned, the value returned by the TIME function will cycle every (2^16)/60 seconds (1092 seconds = 18.2 minutes). Longer timing periods may be measured using the PEEK and POKE features to manipulate the most significant bytes of the real time clock. See programs in Appendix C: Sample Programs for examples.

Example:

```
         >
enter    > PRINT TIME(1)
output     924
         >
```

COSH(expression)	Returns the hyperbolic cosine of the expression.
SINH(expression)	Returns the hyperbolic sine of the expression.
TANH(expression)	Returns the hyperbolic tangent of the expression.
ATAN(expression)	Returns the arctangent of the expression. The range is +PI/2 to -PI/2 radians.
ASIN(expression)	Returns the arcsine of the expression. The range is +PI/2 to -PI/2 radians.
GAMMA(expresion)	Returns the value of the gamma function (the generalized factorial function) for the expression.
FREE(0)	PRINT FREE(0) prints the number of unused bytes available in memory.
MEM (variable name)	Returns the address in memory of the variable given as an argument. This is useful in assembly language CALLs.

6.1.2 Intrinsic Functions Directly Accessing Memory and the 8080 System

(See Appendix E, Interfacing with the Assembler and Memory, for a full explanation of the use of these functions.) Numbers in intrinsic functions must be decimal. Therefore, all hexadecimal numbers must be converted to decimal numbers before using them as arguments in intrinsic functions.

INP(8080 port)	This function allows the programmer to perform an 8080 IN instruction from the specified port. The statement PRINT

INP(80) tells you what value is in the 80th port of the System 88.

OUT 8080 port, expression

This instruction allows the programmer to perform an 8080 OUT instruction to a specified port. For example, OUT 40,3 performs an OUT 40 instruction with 3 in the 8080 accumulator.

POKE memory byte, expression

This function allows the programmer to fill the specified byte in memory with a given expression value. For example, POKE 3000,J+3 will fill memory byte 3000 decimal with the value J+3. This function should be used with caution, since improper use may wipe out portions of the contents of memory.

PEEK(memory byte)

This function allows the programmer to examine the value being held in the specified memory byte location. For example, PRINT PEEK(3000) will tell you what value is in memory byte 3000 decimal.

6.1.3 Intrinsic String Functions

(See Section 7, Strings and Arrays, for a discussion of strings.)

LEN(string variable)

Returns the number of characters in the specified string.

Example:

```
         >
  enter  >10 A3$="PICKLE"\PRINT LEN(A3$)
         >RUN

  output 6
         >
```

VAL(string variable)

Returns the numeric value of a numeric string if the string doesn't contain blanks.

Example:

```
         >
  enter  >PRINT VAL("123")
  output 123
         >
```

STR$(expression)

Returns a string with the specified numeric value.

Example:

```
         >
  enter  >PRINT STR$(234)
  output 234
         >
```

Within the STR$ function it is possible to define the format of the resulting string by using the syntax:

```
STR$(expression,%#format specifiers)
```

For example:

```
        >A=1234
        >PRINT STR$(A,%#$6I)
output  $1234
        >
```

ASC(string variable) Returns the decimal representation of the ASCII code for the first character in the string specified. See Appendix C, The BASIC Character Set, to find the ASCII code in BASIC.

Example:

```
          >
  enter   >S$="S"
          >PRINT ASC (S$)
  output  83
          >
```

CHR$(expression) Returns a string specified by the expression. The expression is a decimal representation of the ASCII code.

Example:

```
          >
  enter   >PRINT CHR$(83)
  output  S
          >
```

LEFT$(expression,n)

Returns the left-most n characters of the string expression. n may also be an expression.

Example:

```
          >
  enter   >A$="HELLO"
          >PRINT LEFT$(A$,2)
  output  HE
          >
```

RIGHT$(expression,n)

Returns the right-most n characters of the string expression. n may also be an expression.

Example:

```
        >
enter   >A$="HELLO"
        >PRINT RIGHT$(A$,2)
output  LO
        >
```

MID$(expression,n,m)

Returns the nth through the mth characters of the string expression, inclusively. Both n and m may be expressions.

Example:

```
        >
enter   >A$="HELLO"
        >PRINT MID$(A$,3,4)
output  LL
        >
```

6.2 USER-DEFINED FUNCTIONS

BASIC allows programmers to define their own single or multi-line functions or one-line functions within a program. The function name begins with the letters FN, followed by a legal string or numeric variable name. If the function is a one line function, the definition takes the form DEF FN legal variable name (arguments)=function. This is a one-line function. For example: DEF FNA1(A,B)=A+B. The arguments of the function (A and B) are local to the function definition. That is, their values are not affected outside of the execution of the function. Therefore, when the function is called upon during program execution, the arguments of the function call are substituted in for the dummy statement of the function definition. For this reason, the number of arguments in the function definition must always equal the number of arguments in the function call, or an error message will be generated.

Example:

```
 enter  >LIST
        10 PRINT "Use control-Y to exit"
        20 DEF FNS1(A,B)=A+B
        30 INPUT1 "Give 2 numbers--",X,Y
        40 PRINT " Their sum is: ",FNS1(X,Y)
        50 PRINT " The absolute value of their sum is: ",ABS(FNS1(X,Y))
        60 GOTO 30
        >RUN

 output Use control-Y to exit
        Give 2 numbers-- 4,-56 Their sum is: -52
         The absolute value of their sum is: 52
        Give 2 numbers-- 34.78,-567 Their sum is: -532.22
         The absolute value of their sum is: 532.22
        Give 2 numbers--  (Control-Y command used here)
        Interruped in line 30
        >>
```

If the user-defined function is a multi-line function, the first line of the function takes the form DEF FN legal variable name (arguments). The lines following that statement form the definition of the function. The last line of the function definition must be the statement FNEND, to indicate the end of the definition. A multi-line definition must return a value. This is done by using a RETURN statement with the variable or constant to be returned. The RETURN statement informs BASIC when executing the function that computation is over.

Example:

```
                >
     enter      >10 DEF FNA (X,Y,Z)
                >20 IF Z=1 THEN RETURN X
                >30 X=Y*Z+X*3
                >40 RETURN X
                >50 FNEND
                >60 A=1\B=2\C=A+B
                >70 PRINT FNA (A,B,C)
                >RUN

     output     9
                >
```

In the example above, note again that the variable names in the function definition are local to that definition; when the definition is called later, the variable names used in the call are completely different from those in the function definition. The function definition and call must only contain the same number and type of variables. Functions must be defined within the program only once, and a definition must exist for each user-defined function called in a program.

6.3 SUBROUTINES

Subroutines are used in much the same way as user-defined functions. Their purpose is to allow the programmer to define a section of the program which may be used again and again during program execution to perform a desired function. The GOSUB statement is used to call the subroutine. Execution of the program is transferred to the program line specified in the GOSUB statement. This line is the beginning of the subroutine. The end of the subroutine is indicated by a RETURN statement. When BASIC encounters a RETURN statement, it returns to the program statement after the GOSUB statement. BASIC then goes on with the rest of the program.

Example:

```
                >
     enter      >10 INPUT1 "Give positive #: ",X
                >20 IF X>0 THEN GOSUB 200 ELSE 10
                >30 REM REST OF PROGRAM
                >40 STOP
                >50 REM Subroutine next
                >200 PRINT " Square root of your"
                >210 PRINT "number is: ",SQRT(X)
                >220 RETURN
                >RUN
```

output
```
Give positive #: 356  Square root of your
number is:  18.867963
Stop in line 40
>>
```

Take care not to let program execution "fall into" the subroutine. For example, in the above program, if you remove the STOP statement at line 40, the subroutine will execute twice-- once when called in the GOSUB statement, and once when BASIC moves on to line 200 from line 30. This situation results in an error message being generated by BASIC, since BASIC finds two RETURN statements but only one GOSUB statement in the program.

Example:

enter
```
>40
>LIST
10 INPUT1 "Give positive #: ",X
20 IF X>0 THEN GOSUB 200 ELSE 10
30 REM Rest of program
50 REM Subroutine next
200 PRINT "  Square root of your"
210 PRINT "number is: ",SQRT(X)
220 RETURN
>RUN
```

output
```
Give positive #: 569.234 Square root of your
number is:  23.858625
 Square root of your
number is:  23.858625

220 RETURN
          ^
RETURN without GOSUB error
>
```

Section 7

STRINGS AND ARRAYS

Two of the more advanced elements of a BASIC program are strings and arrays. They are incorporated into one section in this manual because, in many ways, a string can be treated in the same way as an array.

7.1 ARRAYS

An array is a list of items which may be represented by a legal variable name and indexed by a subscript of that variable. For example, the list (1,2,3,4,5) may be represented by the variable X. The first item in the list would be referenced by subscript 1 (written X(1)). Note that subscripts denoting a position in an array begin with 1. The second item would be referenced by the subscript 2 (X(2)), and so on. The subscripts may in turn be represented by a variable (X(I)).

In using arrays, the user must first dimension the array using the DIM statement. Otherwise there will be an error.

Example:

```
               >
               >LIST
     enter     10 REM Print out array in reverse order
               15 DIM X(5)
               20 X(1)=10\X(2)=20\X(3)=30\X(4)=40\X(5)=50
               30 FOR I=5 TO 1 STEP -1
               40 PRINT X(I)
               50 NEXT
               >RUN

     output    50
               40
               30
               20
               10
               >
```

7.1.1 The DIM statement.

The DIM statement takes the form DIM variable array name (number of items). For example: DIM X(500). An array may be dimensioned only once in a program. An array may contain more than one dimension. For example, the following table is a representation of a 2-dimensional array.

Array X(I,J):	J=	1	2	3	4
	I=1	10	11	12	13
	2	14	15	16	17
	3	18	19	20	21
	4	22	23	24	25

The position X(4,3) contains the number 24. A sample program to print this array would be:

```
        >
enter   >10 DIM X(4,4)
        >20 FOR I=1 TO 4\FOR J=1 TO 4
        >30 READ X(I,J)\PRINT X(I,J),
        >40 NEXT\PRINT
        >50 NEXT
        >60 DATA 10,11,12,13,14,15,16;17,18
        >70 DATA 19,20,21,22,23,24,25
        >RUN

output  10 11 12 13
        14 15 16 17
        18 19 20 21
        22 23 24 25
        >
```

Although we are not able to represent more than two dimensions in this matrix form, more than two dimensions may be assigned to an array. The number of dimensions is limited only by available memory space. Each item in an array takes up five bytes of space.

7.1.2 Optional Array Origins.

The user may, if desired, set his or her arrays to start indexing from 0. To do this, use the statement DIM0 before dimensioning arrays with other DIM statements. (There is also a DIM1 statement, but this is the default that BASIC will automatically assume.)

7.2 STRINGS

A string is a list of characters (which may include blanks) surrounded by quotation marks. If you put anything in quotation marks, BASIC will think it's a string. Quotation marks tell the computer to simply reproduce whatever information is contained within the marks. A string is represented by a string variable, which is any legal variable name, followed by a dollar sign ($) symbol: "A1$."

System 88 BASIC also has true string arrays. They are just like numeric arrays except that each element, instead of being a number, is a string.

An example of the correct way to dimension a string array is:

```
DIM A$(5,5:15)
```

Here we have a two-dimensional array (5 x 5), each element of which is a string that has a MAXIMUM of 15 characters. Note the use of the colon-- it tells BASIC that we are done with the dimensioning and now want to set an upper limit on the length of the string elements of the array.

Example:

```
SCR
10 DIM A$(3:6)
20 A$(1)="red" \ A$(2)="yellow" \ A$(3)="green"
30 PRINT "What color is the traffic light?"
40 PRINT A$(RND(3))
50 PAUSE 120
60 GOTO 30
RUN
```

Notice that we do the calculation of the subscript directly inside the parentheses.

You may use string variable "scalars," without subscripts, but they are limited to twelve characters. In fact, because strings themselves are like arrays, BASIC will treat a string variable "scalar" as an array with dimensions (1). Thus, it is possible to make the string variables longer than twelve characters by dimensioning them that way.

Example:

```
          >
enter     >SCR
          >10 DIM A$(1:25)
          >20 A$="abcdefghijklmnopqrstuvwxy"
          >30 PRINT A$
          >RUN

output    abcdefghijklmnopqrstuvwxy
```

Strings and string variables may be used in combination with LET, READ, DATA, PRINT, IF and INPUT statements. The IF statement does produce alphabetic comparisons when the relational operators are used.

Example:

```
          >
enter     >100 IF Z$+B$<"Smith" THEN 50
          >
```

When string variables are used in an INPUT statement, the input must not be surrounded by quotation marks. When strings occur in DATA statements, they must be surrounded by quotation marks.

You cannot have a string scalar and a string array with the same name. For instance:

```
A$="Hello"
DIM A$(N:M)
```

is an error if N>1.

Section 8

THE MAT STATEMENT

The PolyMorphic disk BASIC MAT statement differs considerably from MAT statements in other BASICs used in many other computers. Readers who are already familiar with other BASICs and other MAT statements will probably be pleasantly surprised.

8.1 MAT

The MAT statement is a general array operator. It may be used with ANY array, numeric, or string, not just matrixes (despite the name).

First, a very simple example. Enter the following program:

```
>SCR
>10 DIM A(100)
>20 MAT A=5
>30 MAT PRINT A,
>RUN
```

This short example shows the two correct syntaxes for the MAT statement. In line 20 we set every element of the array A to the constant 5. However, just as in assignment statements for single variables, the right-hand expression can be extremely complex. In line 30 we printed the array A by combining the MAT statement with a PRINT statement. A number of BASIC statements can be combined with MAT.

8.2 ASSIGNMENTS USING THE MAT STATEMENT

Now we'll try something a bit more complex.

```
>SCR
>10 DIM A(5,5),B(25)
>20 MAT A,B=RND(0)
>30 FOR I=1 TO 5 \ FOR J=1 TO 5
>40 PRINT A(I,J), \ NEXT
>50 PRINT \ NEXT
>60 FOR I=1 TO 25 \ PRINT B(I), \ NEXT
>RUN
```

Here we see four important aspects of the MAT statement. First we notice that it does, in fact, work with multi-dimensioned arrays. Second, we see that each element of the array was set to a DIFFERENT random number. This means that, in effect, the MAT statement was executed once for each element of the array. Thus we can say that the MAT statement is an implied FOR...NEXT loop over all the elements of the specified array (which in this case was A). This is an extremely important point, and we will emphasize it repeatedly. Third, we notice that the multiple assignment capability also applies to MAT statements. And fourth, we notice that MAT works on a row major basis, as can be seen by the fact that A(1,2)=B(2) etc.

8.2.1 Multi-Dimensioned Arrays Using MAT

The MAT statement will accept any array-- of any dimension.

As mentioned above, the MAT statement can be considered an implied FOR...NEXT loop. For example, consider what happens when BASIC is given the following:

```
>SCR
>10 DIM A(10),B(10),C(10)
>20 MAT B=RND(0) \ MAT C=RND(10)
>30 MAT A=B+C
>40 MAT PRINT A,
>RUN
```

We can consider this as producing the same results as the FOR... NEXT loop:

```
....
>30 FOR I=1 TO N
>40 A(I)=B(I)+C(I)
>50 NEXT
>60 MAT PRINT A,
>RUN
```

where N is the number of elements in the array A. But the MAT statement is not only more concise than FOR...NEXT, saving memory space because it makes programs smaller, but it runs much faster-- in the example above, nearly three times faster.

Suppose, however, that you have a scalar variable that you would like to add to an array. It would be nice to be able to write your program like this example:

```
>SCR
>10 DIM A(3,3)
>20 B=RND(10)
>30 MAT A=RND(0)+B
>40 MAT PRINT A
>RUN
```

Fortunately, PolyMorphic disk BASIC will let you.

How does BASIC know whether a variable in a MAT statement is an array or not? Like this: If BASIC "sees" a variable in a MAT statement, and you have dimensioned an array with that name, BASIC will use the array. If there is no array by that name, BASIC will get the scalar variable with that name. If there is neither an array nor a scalar with that name, BASIC gets a 0. We can see this by inserting the following line into the program above:

```
>15 DIM B(9) \ MAT B=99
>RUN
```

Remember that you can make the expression as complex as you wish. For instance, this program

Examples

8.2.1

I

4.15625 1.5239258 2.2120819 5.9189758
6.6647797 2.6861572 1.8344269 6.740448
9.81142

II

ϕ ϕ ϕ ϕ ϕ ϕ ϕ ϕ ϕ ϕ

III

2.5239258
2.2120819
2.8190765
2.9189758
2.6647797
2.6861572
2.8344269
2.740448
2.81142.

IV

99.523926
99.212082
99.819077
99.918976
99.66478
99.686157
99.834427
99.740448
99.81142

```
>SCR
>10 DIM A(3,3)
>20 MAT A=RND(0)
>30 MAT A=ASIN(A)
>40 MAT PRINT A
>RUN
```

calculates the ARCSIN of every element of A. Of course, the program would be smaller if we wrote:

```
>20 MAT A=ASIN(RND(0))
>DEL 30
>RUN
```

Or suppose that we have arrays of X and Y coordinates of points and would like to know the distances to the origin:

```
>SCR
>10 DIM X(10),Y(10),D(10)
>REM All the points lie within the unit circle
>20 MAT X=1-2*RND(0) \ MAT Y=1-2*RND(0)
>30 MAT D=SQRT(X*X+Y*Y)
>40 MAT PRINT X,Y,D
>RUN
```

8.2.2 Multiple Assignments Using MAT.

In the same way that you use multiple assignments with scalar variables, you can write BASIC statements using MAT for multiple assignments. (Note: If you are going to assign values to several arrays by using one multiple assignment MAT statement, make sure that you dimension all of those arrays to the same length. Not doing so may result in unpredictable behavior on the part of BASIC.) For example, suppose we want to reset a number of arrays to 0:

```
>10 DIM A(100),B(100),C(100),D(100)
...
>REM   Here we have some random program
...
>950 MAT A,B,C,D=0
```

Line 950 is thus equivalent to the FOR...NEXT loop:

```
>950 FOR I=1 TO 100
>951 A(I),B(I),C(I),D(I)=0
>952 NEXT
```

8.2.3 Order of Assignment in MAT Statements

Here it will help to remember the implied FOR...NEXT loop. Consider this program:

```
>SCR
>10 DIM A(2,2)
>20 MAT A=RND(0)
```

It is clear that the first element of A to be assigned a random number is A(1,1). But is the next random number assigned to A(1,2) or to A(2,1)? As we saw in an earlier example, the next random number goes to A(1,2). This is a row-major procedure. The right-most dimension varies most rapidly.

The implied FOR...NEXT will also help in the next example:

```
>10 DIM A(10),B(10)
>20 MAT A,B=RND(0)
```

The first variable assigned is B(1). (Remember that multiple assignments proceed right-to-left.) Using the idea of an implied FOR...NEXT, we can see that the subsequent assignments go to A(1), B(2), A(2), B(3), A(3) ....

8.3 MAT IN COMBINATION WITH OTHER BASIC STATEMENTS

As we saw in the first example, MAT can be combined with other BASIC statements. Below is a list of the statements that can be combined with MAT:

```
LET
PRINT
READ
INPUT
PLOT
IF..THEN..ELSE
```

We have already seen how to make assignments with the MAT statement. Also, from the examples above using MAT PRINT one can see that they are straightforward examples of the principle of an implied FOR...NEXT loop. In exactly the same way, one can use MAT READ, MAT INPUT, and MAT PLOT.

```
>SCR
>10 DIM A(10),B(10)
>20 MAT READ A
>30 MAT PRINT A, \ PRINT
>40 MAT INPUT B
>50 MAT PRINT B, \ PRINT
>60 PAUSE 60
>70 PRINT CHR$(12) \ PLOT 0,44,0
>80 MAT PLOT A,B,1
>90 DATA 10,20,30,40,50,60,70,80,90,100
>RUN
```

8.3.1 MAT IF Statements

This is one of the most powerful uses of the MAT statement. Remembering the principle of implied FOR...NEXT, we see that:

```
>SCR
>10 DIM A(10)
>20 MAT A=RND(10)
>30 MAT IF A=0 THEN 50
>40 PRINT "No element of A equals 0" \ STOP
>50 PRINT "Some element of A equals 0"
>RUN
```

is essentially the same as:

```
>SCR
>10 DIM A(10)
>20 FOR I=1 TO 10 \ A(I)=RND(10) \ NEXT
>30 FOR I=1 TO 10
>40 IF A(I)=0 THEN EXIT 70
>50 NEXT
>60 PRINT "No element of A equals 0" \ STOP
>70 PRINT "Some element of A equals 0"
>RUN
```

Notice that the MAT statement simulates the EXIT feature. Thus when we jump out of the MAT IF we do so in the best way-- BASIC doesn't blow up on us!

The example above illustrates perhaps the most useful feature of MAT IF. It allows one to test an entire array for a certain condition and branch if it is met. We also can do this:

```
>SCR
>10 DIM A(100)
>20 MAT A=RND(10)
>30 MAT IF A=2*INT(A/2) THEN PRINT A," is even"
>RUN
```

which prints every even element of the array A.

8.4 THE # FEATURE IN MAT STATEMENTS

Sometimes when using an IF...THEN inside a FOR...NEXT loop we are interested in knowing which element of the array met the condition that caused us to branch out. How, one should ask, can we do that with a MAT IF?

PolyMorphic disk BASIC has a variable # that is used as the index of the implied FOR...NEXT loop that a MAT statement is simulating. For example:

Prints all numbers from 1 to 100

```
>SCR
>10 DIM A(100)
>20 MAT A=#
>30 MAT PRINT A,
>RUN
```

We see that the array is now all the integers from 1 to 100. The # variable may be freely used in the right-hand expression:

```
>SCR
>10 DIM A(90)
>20 K=PI/180 \ REM convert degrees to radians
>30 MAT A=SIN(#*K)
>40 MAT PRINT A," is the sine of ",%3I#," degrees"
>RUN
```

Here we print out a table of the sines of the angles from 1 degree through 90 degrees.

How does this pertain to MAT IF?

Well, when we leave the MAT IF, the variable # will be set to the index ot the array element that met the condition. For example:

```
>SCR
>10 DIM A(100)
>20 MAT A=RND(0)
>30 MAT IF A>.95 THEN 50
>40 PRINT "no luck this time" \ STOP
>50 PRINT A(#)," is greater than 0.95"
>RUN
```

If we wanted to print every element of A greater than 0.95, along with their indexes, we would change the program:

```
>30 MAT IF A>.95 THEN PRINT A,#
>DEL40,50
>RUN
```

What happens with multi-dimensioned arrays? Well, since the # is a single variable, it can't tell us the indices for each dimension. Instead, it treats the array as a vector:

```
>SCR
>10 DIM A(10,10)
>20 MAT A=#
>30 FOR I=1 TO 10 \ FOR J=1 TO 10
>40 PRINT A(I,J), \ NEXT
>50 PRINT \ NEXT
>RUN
```

What happens when the arrays have been set to a base of 0 by the DIM0 statement? BASIC will start the # "index" from 0 instead of from 1.

```
>SCR
>10 DIM0
>20 DIM A(10)
>30 MAT A=#
>40 MAT PRINT A,
>RUN
```

8.5 MAT WITH STRING ARRAYS

As we mentioned earlier, the MAT statement will accept string arrays. Let's re-write an earlier program to see how it works.

```
>SCR
>10 DIM A$(90:25)
>20 MAT A$="The sine of "+STR$(#)+" degrees is"
>30 K=PI/180
>40 MAT PRINT A$,SIN(#*K)
>RUN
```

This example shows that MAT statement handles string functions for string arrays in the same way it handles numeric functions for numeric arrays. Note that the # feature is used! We leave as an exercise to the reader the task of writing the above program without MAT (just to convince yourself how useful MAT is).

8.6 SPECIAL ARRAY FUNCTIONS

There are six intrinsic array functions in PolyMorphic disk BASIC. Like the MAT statement, they can be considered implied FOR...NEXT loops. But they are very, very much faster than the corresponding loop, a fact you can test for yourself.

Unlike the other BASIC functions that take expressions as arguments, these functions take the NAME OF AN ARRAY as the argument.

8.6.1 SUM

This function returns the arithmetic sum of the elements of the array given as an argument:

```
>SCR
>10 DIM A(100)
>20 MAT A=#
>30 PRINT SUM(A)
>RUN
```

8.6.2 PROD

This function returns the product of the elements of the array given as the argument:

```
>SCR
>10 DIM A(10)
>20 MAT A=RND(0)
>30 PRINT PROD(A)
>RUN
```

8.6.3 MAX and MIN

These functions return the maximum and minimum, repectively, of the array given as the argument:

```
>SCR
>10 DIM A(100)
>20 MAT A=RND(0)
>30 PRINT MAX(A),MIN(A)
>RUN
```

Both the MAX and MIN functions modify the indexing variable #. They set # equal to the index of the element of the array which is the maximum (or minimum).

Example:

```
>SCR
>10 DIM A(10)
>20 MAT A=RND(100)
>30 PRINT MAX(A) \ PRINT # \ PRINT A(#)
>40 PRINT MIN(A) \ PRINT # \ PRINT A(#)
>RUN
```

8.6.4 MEAN

This function returns the mean (the average, in usual parlance) of the elements of the argument array.

```
>SCR
>10 DIM A(100)
>20 MAT A=RND(0)
>30 PRINT MEAN(A)
>RUN
```

8.6.5 STD

This function returns the STandard Deviation of the elements of the argument array.

```
>SCR
>10 DIM A(100)
>20 MAT A=RND(0)
>30 PRINT STD(A)
>RUN
```

Compare these functions with the FOR..NEXT loops you would otherwise have to use, and see how much faster these function. For the SUM function, the loop would be:

```
>SCR
>10 DIM A(100)
>20 MAT A=#
>30 FOR I=1 TO 100 \ X=X+A(I) \ NEXT
>40 PRINT X
>RUN
```

Section 9

Max
PLOT 127, 47

THE PLOT FEATURE

The PLOT statement allows the BASIC programmer to use graphics characters to display data. The statement plots data on the video screen on a 128 by 48 grid. The "origin" of the display grid is the lower left hand corner of the screen, addressed as point (Ø,Ø). The X-axis of the grid runs horizontally across the display (left to right) from Ø to 127, and the Y-axis of the grid runs vertically up the display (bottom to top) from Ø to 47.

To plot data using the PLOT statement, use the following form:

```
PLOT X,Y,Z
```

The X is any user-selected variable or expression chosen as the X-coordinate of the plot and Y is the Y-coordinate of the plot. Z is an arbitrary expression-- it will plot the point as a bright spot if Z is odd, and as a dark spot if Z is even. The X-coordinate and Y-coordinate must reference points which are actually on the display grid; for this reason, they must be greater than Ø. In addition, X must be less than or equal to 127, and Y must be less than or equal to 47.

After a point is plotted, the cursor position moves to that point of the screen. The next PRINT or INPUT statement will then appear at that spot. This is useful for arranging input prompts on the screen and for formatting output text.

For faster plotting, PolyMorphic Systems' disk BASIC has another plot feature that lets you plot an entire array at a single stroke. The syntax is:

```
PLOT:A
```

where A is the name of the array to be plotted. It should be dimensioned (n,3), where n is the number of points to be plotted; (i,1) is the X-coordinate of the ith point, (i,2) is the Y-coordinate of the ith point, and (i,3) is the Z-coordinate of the ith point. If the program is running with Ø-based arrays (i.e. if you have executed a DIMØ statement), the array should be dimensioned (n,2).

The statement

```
PLOT:n,m
```

will plot the first m points of array n.

For a demonstration of the PLOT feature, see Appendix C: Sample Programs.

Section 10

OPTIMIZING YOUR BASIC PROGRAM

This section gives you some techniques for optimizing BASIC programs: making programs run faster and reducing the amount of memory they require. Many of the techniques described here reduce execution time as well as the amount of memory used by a program. The sample program at the end of this section also shows you how to time program execution using the real-time clock and how to develop these techniques further.

The first technique is to eliminate extraneous program material. Remove the keyword LET from all assignment statements, since it is not needed. Once the program is running correctly, remove all REM statements; they take up memory space and must be skipped over during program execution, thus increasing execution time. Remove variable names from NEXT statements, since they increase loop processing overhead.

The second technique is to pack as much on a program line as possible. Placing two statements on the same line, rather than on two separate lines, saves three bytes of memory; each line in memory is composed of a count byte, two bytes for the line number, the actual program information, and a carriage return (one byte). Four bytes making up the count, line number, and carriage return are "traded" for the statement separator \ (one byte) when two ines are combined.

Redundant or trivial computation should be removed from FOR-NEXT loops and from statements that are repeatedly executed. For example, the expression 63488+5*64 contains all constants, and may be reduced to the single constant 63808, eliminating the addition and multiplication as well as the overhead of converting the string of characters 63488, 5, and 64 to numeric form for performing the operation. If a program repeatedly uses a constant such as 63488, it is wise to assign that constant to a variable for two reasons: it is faster for BASIC to look up the value of a variable than to convert the string of characters to a number each time; if a commonly used number in the program must be changed, it need only be changed in a single place.

In general, when trying to reduce the amount of memory a program uses, eliminate everything that is not essential-- comments, unneeded blanks, etc. In PolyMorphic Systems Disk BASIC, all blanks can be eliminated.

When trying to reduce the execution time of a program, first find out where the program spends most of its time-- rewriting a section of a program to make it ten times faster will not yield noticeable results if that section of the program is used only 3% of the time. When you have identified the heavily used sections, you can be confident that optimization will make an important difference. It should be noted that an undebugged, untested, or incomplete program is not a good candidate for optimization, since most of the steps outlined above reduce the ease of comprehension of a program and increase the difficulty of finding bugs.

Example (this example is similar to the sample program TIMER in Appendix C):

enter

```
>100 REM Generate timing information for BASIC programs
>110 REM Calculate average timing over 100 samples
>120 REM First calculate loop overhead for 100 iterations
>130 T=TIME(0)
>140 FOR I=1 TO 100
>150 NEXT
>160 T=TIME(1) \ REM Time for 100 iterations
>170 PRINT "Loop overhead is about",T/(100*60),"sec per iteration"
>180 T1=T\ REM Save the overhead time
>190 REM Now time overhead when we use "NEXT I"
>200 T=TIME(1)
>210 FOR I=1 TO 100
>220 NEXT I
>230 T=TIME(1)
>240 PRINT "versus",T/(100*60),"sec per iteration for NEXT I."
>250 REM Now time A=300
>260 T=TIME(0)
>270 FOR I=1 TO 100
>280 A=300
>290 NEXT
>300 T=TIME(1)-T1 \ Rem subtract overhead to get stmt time
>310 PRINT "A=300 takes about",T/(100*60),"seconds to do."
>320 REM Now set B=300, do A=B 100 times
>330 B=300
>340 T=TIME(0)
>350 FOR I=1 TO 100
>360 A=B
>370 NEXT
>380 T=TIME(1)-T1 \ REM Again, subtract loop overhead
>390 PRINT "A=B, for B=300, takes about",T/(100*60)," seconds."
>RUN
```

output

```
Loop overhead is about .002 sec per iteration
versus 2.8333333e-03 sec per iteration for NEXT I.
A=300 takes about 3.1666667E-03 seconds to do.
A=B, for B=300, takes about 2.8333333E-03 seconds.
>
```

Section 11

DEBUGGING BASIC PROGRAMS

System 88 BASIC has a number of useful features that help a programmer debug his programs quickly. This chapter describes these features and illustrates them with sample programs.

11.1 RUN-TIME-ENVIRONMENT

First it is necessary to understand the concept of a run-time-environment. When you are writing a program in BASIC, the variables and arrays that are part of your program do not exist in the computer's memory; they are merely symbols in your program. When you RUN your program, BASIC generates a run-time-environment that includes all the arrays, variables, and user-defined functions in your program. This run-time-environment (RTE) is constantly being changed by BASIC as your program runs.

When for any reason (an error, a STOP statement, or a Control-Y) program execution stops, the RTE is usually preserved. This is why it is possible to continue (CON) after a STOP or Control-Y. PolyMorphic's Disk BASIC has the added feature that the RTE is preserved during line editing. It is not destroyed unless you re-RUN the program from the beginning. Note, however, that a newly created user-defined function will not be recognized unless the program is re-run from the start.

Let's look at an example:

```
>SCR
>100 PRINT "Show RTE SAVE feature"
>200 A=RND(101)
>300 A=A/B
>400 B=100
>500 PRINT A
>RUN
```

As you would expect, BASIC points out the division-by-zero in line 300. What we would LIKE to do to fix this is change the program by...

```
>300 B=100
>400 A=A/B
```

and then RUN from line 300:

```
>RUN 300
```

You notice that the program now works correctly. One might ask at this point, "Why bother to run from line 300? Just RUN from the beginning." This will, of course, work too. But suppose that between line 200 and line 300 there is more of the program that takes ten minutes to execute. Do you really want to wait for those ten minutes to make sure that the changes you have made will work?

There are a few cautions to observe when trying to re-RUN using the saved RTE.

1) You can't RUN from the middle of a FOR...NEXT loop.

2) If the error occurred in a MAT statement that uses the target array in the right-hand expression (for instance: MAT A=SQRT(A)), you must go back to where the array was previously calculated.

3) Be careful that the program doesn't try to re-dimension any of its arrays by executing the same DIM statement.

11.2 THE DUMP COMMAND/STATEMENT

Sometimes the changes you must make are extensive, and it will not always be clear where you need to re-RUN from. In that case, the DUMP command can be used to study the state of the RTE.

DUMP may be used in direct mode to study the RTE, or it may be invoked by a program to display the RTE during program execution.

11.2.1 DUMP in Direct Mode

Whenever BASIC is given the command to DUMP, it displays on the video monitor the values of all the variables currently defined in the RTE. All of the variables are printed in format 12E4, so as to provide a columnar display.

> NOTE: it is also possible to DUMP to BASIC files using the syntax DUMP:N. If the file N is defined as the printer channel, BASIC will DUMP to a printer. We will comment on this again (see Part II, Section 15).

If there are more than 59 variables in the RTE, BASIC will put a series of decimal points on the screen and wait for the user to strike the keyboard before displaying the next 59 variables. (When DUMPing to a printer or disk file, this waiting mode is ignored.)

11.2.2 DUMP as a program statement

In addition to the direct mode, DUMP may be used as a program statement. This is particulary useful when debugging programs with numerous FOR...NEXT loops. The example below shows in general how this can be used.

enter:

```
>SCR
>10 FOR I=1 TO 25
>20 A=RND(0)
>30 S=SIN(A) \ C=COS(A) \ T=TAN(A)
>40 S1=SINH(A) \ C1=COSH(A) \ T1=TANH(A)
>50 REM .. lots of other calculations
>60 REM .. that we won't detail here
```

```
>100 DUMP
>110 NEXT
>RUN
```

Since this is a working program, it merely illustrates the way in which the DUMP statement presents the RTE. If you find that the display does not last long enough, you have two options. To illustrate:

```
>100 DUMP \ WAIT
>RUN
```

Now the display hangs around until you strike a key. Alternately:

```
>100 DUMP \ PAUSE 60
>RUN
```

This time the display stays for one second before the program continues execution. Frequently the FOR...NEXT loop will take enough time by itself for you to study the displayed RTE. Simply putting the DUMP statement at strategic points and watching the monitor can give an excellent idea of how a program is working.

The reader is strongly advised to take a few moments now and experiment with the DUMP statement. Once you are familiar with it, you will find that debugging time can be decreased by as much as ten times (we have had that experience at PolyMorphic), especially when combined with XREF.

Note that DUMPing to a printer or to a disk file during program execution can give you a permanent (and complete) history of the RTE. For complex programs this can be very helpful. (See Part II, Section 15.)

11.3 CROSS REFERENCE (XREF)

System 88 BASIC provides the user with the option of cross-referencing his variables with the line numbers of his program in which they appear. XREF is NOT a program statement!

Just as in the DUMP command/statement, it is possible to XREF to disk files or a printer. In fact, this is what we recommend. We suggest that you enter a program, one of your own or one provided with the system disks, and then do a XREF. At least once, XREF to the video monitor, and judge for yourself the speed of this utility.

11.3.1 Limited XREF

In addition to the global cross-reference provided by XREF, it is possible to limit the cross-reference to specified variables. This is particularly useful in debugging when one discovers that a certain variable (say F7) is incorrectly calculated. By typing

```
XREF F7
```

and then listing the indicated lines, you can easily find the problem area and make appropriate corrections. The user may specify any number of variables in the list after XREF, separating them with commas.

XREF does not provide cross-referencing for user-defined functions.

11.4 SINGLE-STEPPING IN BASIC

System 88 BASIC lets you single-step through your programs one line at a time. You do this with the WALK command, which has a syntax exactly like that of RUN (it resets or preserves the RTE in the same way).

When single-stepping through a program, BASIC will, before executing a line of the program, list the line about to executed. After displaying that line, BASIC waits for input from the keyboard. You have three options:

1. Type X. This continues the single-step mode.
2. Type G. Go: This terminates the single-step mode and RUNs from the current line in the program.
3. Type D. This executes a DUMP command and then single-steps.

Any other keystrike will be ignored.

11.5 ON ERROR

BASIC provides you with optional control over error-recovery with the ON ERROR statement. The syntax is:

ON ERROR statement

where the statement may be any BASIC statement.

You may have any number of ON ERROR statements in the program that you wish. When an error occurs, BASIC will go to the ON ERROR statement that was last executed and execute the instructions found after the ON ERROR statement.

Example:

```
>SCR
>10 ON ERROR GOTO 40
>20 A=10 \ B=5
>30 A=A-1 \ PRINT B/A \ GOTO 30
>40 PRINT " OOPS!"
>RUN
```

Usually you will want the ON ERROR instructions to branch to another part of the program; if it does not, BASIC will continue with the next sequential line after the ON ERROR. Thus if we change the above program:

```
>10 ON ERROR PRINT " OOPS!"
>RUN
```

we are in an endless loop (hit Control-Y!).

Whenever an error occurs, the variable ERR is set equal to the internal code. This allows the user to check for the occurrence of particular errors.

For example, we can change the previous example:

```
>10 ON ERROR PRINT ERR,\GOTO 40
```

and the error code for division-by-zero will be displayed. (See Appendix A for the BASIC error message codes.) Knowing the error message codes can be useful. For example:

```
IF ERR=1036 THEN PRINT "I can't divide by zero; change A."
```

11.6 ON ESCAPE

BASIC also gives the user program control over the escape sequence with Control-Y. The syntax is similar to ON ERROR:

```
ON ESCAPE statement
```

Its use is similar to that of sense switches on older computers. To illustrate its use:

```
>SCR
>10 ON ESCAPE GOTO 100
>20 PRINT "User-controlled escape sequence"
>30 PAUSE 600
>40 REM  UNLESS CTL-Y IS HIT WITHIN TEN
>45 REM  SECONDS WE CONTINUE WITH
>50 REM  "You didn't try to escape!"
>60 PRINT\GOTO 20
>80 REM  BUT IF YOU HIT CTL-Y...
>100 PRINT "You tried to get away, but I gotcha!"
>110 ON ESCAPE GOTO 150
>120 PRINT \ GOTO 20
>150 PRINT "You got away!" \ STOP
```

PART II

BASIC AND THE DISK SYSTEM

The first part of this manual introduced you to the BASIC language. With the information in Part I and in Appendix B, Running BASIC and Loading and Saving BASIC Programs, you can create and save complete BASIC programs. However, that information is only a part of the information that you need to make full use of your disk system.

At this point you can transfer data between your BASIC programs and the monitor screen or the keyboard. Part II explains how you connect BASIC to the other parts of the disk system: the printer, disk files, and special devices. You will learn about file channels, the pathways through which information passes between BASIC programs and disk files, the printer, and the video screen. You will learn how to create BASIC data files and how to pass information in and out of them using file channels. If you are unfamiliar with the idea of disk files, see the System 88 User's Manual for more information.

Using disk files and a printer from within BASIC greatly expands the range and versatility of your programs. You will now be able to save and print the data generated by your programs, and to access that data file from any BASIC program. The kinds of things you might have in a BASIC data file include a mailing list of customer addresses, a list of invoice numbers, a series of square roots, a list of part numbers in your inventory, a list of points to be plotted, etc.

Section 12

FILE CHANNELS

File channels are pathways used by the system to transfer data between different parts of the system. You can think of a file channel as a wire that you hook up between a disk file and your BASIC program. Data flows from the file to the program or from the program to the file. You "hook up" or assign these file channels to a disk file, a printer, the video board, the keyboard, or a special device by way of a FILE statement (Section 14), which we'll discuss later. Once you attach a file channel to a device or disk file, you can output information from that file or put information into that file from within your BASIC program. To do that you use the BASIC file-handling commands (Section 14). We will discuss how to send information to a printer in Section 15.

12.1 THE BASIC FILE CHANNELS

In BASIC there are eight file channels, numbered 0 through 7. Some of these eight channels are permanently allocated for particular system use. Channel 0 is for inputting data from the system keyboard; Channel 1 is for outputting data to the video screen. Channels 2 and 3 are reserved for outputting data to a printer or to a special device. These four channels (0-3) cannot be used for disk files. Channels 4-7 may be used for disk files <u>or</u> for a printer or a special device.

Up to four channels may be assigned at one time for disk file data transfer, plus one channel for printer access and one channel for a special device. Because you can use file channels 4 through 7 for transferring data between a BASIC program and data files, you can use up to four data files at a time within a BASIC program. Each data file can have only one file channel assigned to it at a time. If you are writing information into data files, you cannot have more than one of those files in use on the same disk at the same time; they must be on separate disks. If you are reading information from data files, all of the four possible files may be on the same disk.

12.2 FILE CHANNEL MEMORY USE

We will be discussing how to use file channels in a later section. However, it is important to note before you begin to use file channels that assigning a file channel to a disk file USES UP MEMORY. This may or may not be important, depending on how much memory you have on your system and how long your BASIC programs are. However, to avoid memory-full errors, it is wise to use as few file channels as you can.

Each active disk file uses a 300-byte buffer in memory. (A buffer is a working space set aside in memory.) The first time a file channel is opened to a disk file in a BASIC program, this buffer area is allocated from the space available in

BASIC. Even if you close that channel by using a CLOSE statement, that buffer area is not assigned to another channel. The buffer area remains assigned to that file channel unless you use a CLEAR or SCRATCH command or leave BASIC using the BYE command. This means that once you assign a file channel in a BASIC program, 300 bytes (a byte is a small unit of memory equal to eight "bits") is effectively gone from the area of memory that you can use. Since there is a maximum of four disk files in use in BASIC at one time, up to 1200 bytes may be used up by these disk file buffers. If there is not enough memory to allocate a buffer for a file channel, a memory-full error message is generated.

To save memory, use only as many file channels within a BASIC program as you really need. After you CLOSE a file, re-use the file channel buffer by assigning that file channel to a different file instead of using a new file channel. To see how many bytes of memory are free, use the BASIC direct statement:

```
PRINT FREE(0)
```

Section 13

BASIC DATA FILES: OVERVIEW

Until now we have talked about BASIC programs that calculate values and display words and plot graphs on the screen. But we have not seen a method for PRESERVING the data calculated or displayed by a BASIC program. You might want to calculate the first twenty prime numbers. How do you keep a permanent record of those numbers? You can do so by having your BASIC program send that data out either to a printer or to a file on the disk-- a BASIC data file. Both of these methods require that you use file channels. The following section discusses BASIC data files. Section 15 tells you how to use a printer from within BASIC.

A BASIC data file is a disk file that you build from within a BASIC program. A BASIC program can read from, as well as write into, a BASIC data file. The data file holds information that your BASIC program generates or uses. When we talk about data in this context, we are talking about any information that BASIC can read or write. This means words <u>or</u> numbers. Using the BASIC file handling commands, you can place lines of text into a data file as well as lists of numbers.

Using the various file handling commands, you can create a data file, open it, place data into it; and close it again. Then at a later time you can open the file and read information from it. This entire process is handled by BASIC file handling commands which appear as program statements in your BASIC program. You are already practiced at inputting data from the keyboard (by using INPUT or INPUT1) and displaying data on the screen (by using PRINT or PRINT,). The only difference in using a data file is that you are inputting from a file instead of a keyboard and outputting to a file instead of to the screen.

Section 14 tells you about the various BASIC file-handling commands to use when creating and using data files. The next few paragraphs give an overview of what occurs when you create a BASIC data file and when you open it for use. This is a very general explanation; for the details see Section 14.

Before your BASIC program uses a data file, you must tell BASIC that you are going to use that file. If the file does not already exist, you must create the file. You must also tell BASIC which file channel you want to assign to that data file for data transfer to and from the file. In addition, you must tell BASIC whether you want to read from the file or write to it. When you are done transferring data, you must close the file. All of the above functions are performed using the FILE statement (Section 14.1). After you are ready to use the file, you use various file input and output statements to transfer the actual data over the assigned file channel (see Section 14.4). When you are completely finished writing data to a file, you close that file (Section 14.5).

An important point to remember when using data files is that at some point in the future you are going to want to retrieve the information that you have placed in that file. Make sure that you know whether the type of data you are writing into a file is numerical or string, so that when you access that data again you will know whether to input a string or numerical variable. You can see that trying to input a string like "WORD" from a data file by inputting a numerical variable would result in an error since BASIC is looking for a number.

13.1 DATA RECORDS

The data that you write into a file is arranged in groups called data records. A data record is simply the characters between two carriage return symbols. As you write data into a data file, BASIC forms data records in that file. Each data record has a number; the first data record is record #1, the second data record is record #2, and so on. Section 14.4 tells you how to place data into data files and how to read data from them. You will also learn more about how those procedures form data records.

Section 14

CREATING AND USING BASIC DATA FILES

In Section 13 we discussed the idea of a BASIC data file. Section 14 tells you how to actually build and use a BASIC data file. Remember that all of the statements we discuss below are actual program statements and as such appear on program lines along with the rest of the statements in your BASIC program. For examples of programs that use the file-handling commands, see Section 16.

14.1 FILE STATEMENT ELEMENTS

The most important part of the BASIC file-handling process is the use of the FILE statement. When you use a FILE statement you tell BASIC that you are going to use a data file. Whenever you use a data file in BASIC, you MUST use a FILE statement in your program first to tell BASIC which data file you want to use (or if the file does not yet exist, to tell BASIC that you want to create a data file).

A FILE statement always begins with the word FILE followed by a colon and a number (a file channel number). Then follows a list of FILE statement elements (the particular elements depend upon the function of that particular FILE statement). The form of a FILE statement depends upon its use, but a typical FILE statement follows this form:

```
FILE:n,keyword,file specification,file mode
```

EXAMPLE:

```
FILE:6,OPEN,"<2>Real-Estate",INPUT
```

The various elements of the FILE statement tell BASIC: 1) which file channel the system is going to use to transfer data to or from the disk file-- channel 6 in the example above; 2) the action we are going to perform on the data file (open, close, rewind, etc.); 3) the name of the data file to use or create; and, 4) whether we are reading from (input) or writing to (output) the data file. Note that the terms input and output are used from the point of view of the central processor: input moves from the file to the processor, output from the processor to the file.

Before we talk in more detail about the elements of a FILE statment, take a look at what some typical FILE statements look like:

```
FILE:6,OPEN,"<2>DATA.F2",OUT

FILE:5,CLOSE

FILE:7,POS,23

FILE:U,POS,I-4

FILE:4,REW

FILE:5,OPEN,F$,INOUT
```

Let's take a brief look at the possible elements of a FILE statement. When we list examples of the element, we also tell you where to find a description of that item in the manual.

14.1.1 The File Channel

We've already discussed file channels (Section 12), the data transfer pathways that link data files (and other things, such as a printer) to a BASIC program. You'll recall that there are eight such channels, numbered 0 through 7.

Every BASIC file-handling command (including the FILE statement) includes a file channel number as part of the command. This tells the system which channel to use for transferring data. Whenever you open a data file, you assign a file channel number to it. From then on within the program, every time you transfer data over that channel, the data goes between your BASIC program and the data file assigned to that channel.

The file channel (represented by the symbol n in the rest of this section) may be any variable (a symbol representing a number) or expression evaluating to a correct file channel number. (An expression is a mathematical term or terms separated by arithmetic operator symbols--e.g., 2+(45.6/SIN(A)).) BASIC always evaluates an expression (that is, reduces it to its most basic value). You might begin a FILE statement thus:

```
FILE:4,file statement elements
```

You could select the <u>same</u> file channel number by saying:

```
FILE:SQRT(16),file statement elements
```

14.1.2 The Keyword

The keyword in the FILE statement tells BASIC what action is going to be performed on the data file. Using the keyword, we can create a new data file, open an old one for input, or position to the beginning (or to any particular data record) of an existing file. We can also close a data file to output.

(Remember that when we talk about using a data file for input we are talking about bringing information IN from that file; when we talk about using a data file for output, we are talking about writing information OUT to that file.)

Keyword:	For Information See:
OPEN	Section 14.2, Creating a Data File Section 14.3, Opening a Data File for Input
POS	Section 14.6, Selecting a Particular Data Record
REW	Section 14.6, Selecting a Particular Data Record
CLOSE	Section 14.5, Closing a File to Output

We'll discuss a special keyword, DEF, in Section 15. That keyword allows you to hook up special devices to BASIC using your own machine language file-handling routines. LIST lets you send data to a printer; see Section 15 for information.

14.1.3 File Specification

A file specification is a string or string variable that tells BASIC the name of the data file to create or use. The file specification must be a legal file specification and include a disk specifier unless the file is on the System Drive. (See the System 88 User's Manual for information on file specifications.)

Only FILE statements opening or creating a file use file specifications (that is, only FILE statements with the keyword OPEN). In all other cases, a data file has already been assigned to the file channel given in the FILE statement, and so it need not be named.

Remember that a string is a group of characters enclosed by quotation marks. A string variable is a variable which represents a string. (A string variable takes the form of a legal BASIC numerical variable followed by a dollar sign-- e.g., A$, F1$, etc.) The string variable must represent a legal file name.

EXAMPLES:

```
10 FILE:6,OPEN,"<2>DATA-FILE",OUT

10 F$="<3>LIST/1"
20 FILE:2+4,OPEN,F$,INPUT
```

Note: BASIC automatically dimensions a string to only ten characters unless a DIM (dimension) statement is used within your program to reserve more room in memory for the string. Make sure that you dimension your string to the proper length. If your string is longer than the length reserved for it, BASIC will shorten your string to fit.

14.1.4 The File Mode

The file mode tells BASIC in which direction we want to transfer data: from the program to the file, or vice versa. We can even do both, using the INOUT mode. In INOUT mode we read data, update that data, and write it back out again to the data file.

File Mode	For Information See:
OUT	Section 14.2, Creating a Data File
INPUT	Section 14.3, Opening a Data File for Input
INOUT	Section 14.7, Updating Data Records

14.2 CREATING A DATA FILE: OPEN Keyword and OUT File Mode

You cannot write into a file that already exists. If you try to do so, the system will tell you: That channel not open for output. (The one exception to this is when you are using a file in INOUT mode, which allows you to update data records. See Section 14.7, Updating Data Records.) If you are going to write into a file, you must create that file from within your BASIC program by using a FILE statement. Construct your FILE statement in the following way:

1) Choose a file channel to assign to the file you are creating. Make sure that the file channel is not assigned to any other open file. Use only file channels 4-7.

2) Use the OPEN keyword. This tells BASIC that you are assigning the file channel you have chosen to the data file you are creating. This keyword also tell BASIC that you are "opening" the file; that is, that you will be transferring data between it and the BASIC program.

3) Choose a legal file specification, enclosing it within quotation marks to make it a string. Either place the file specification in the FILE statement directly, or use a string variable in the FILE statement which represents that string. If you have already created a data file and it will be open at the same time as your new file, make sure that the two files are not on the same disk.

   If you do not specify a disk extension, BASIC will automatically affix the default data file extension, .DT, when it creates your data file.

4) Use the OUT file mode. This tells BASIC that you are going to be writing data out from the BASIC program into the data file, and therefore that you are creating the file.

5) From this point on, output data to the file using the PRINT or OUT commands (Section 14.4).

6) Close the file using the CLOSE command (Section 14.5).

A short example of a BASIC program that creates two data files follows. Remember to dimension any strings to their proper length; BASIC automatically dimensions strings to only ten characters.

```
10 DIM A1$(1:20)
20 FILE:5,OPEN,"<2>Inventory",OUT
30 A1$="<3>Mailing-List"
40 FILE:4,OPEN,A1$,OUT
```

For information on the file-handling commands that actually transfer data between a BASIC program and data files, see Section 14.4. Any time you open a file for output, BASIC is pointing to the beginning of the data file; when you print information to the file, you form the first data record, record #1.

Marking the End of the File:

Later, when you input data from the file that you are creating, you will want to be able to tell when you are at the end of the data file. Otherwise you might try to keep on inputting data that does not exist. One way to do this is to remember to put a special character or number at the end of your file when you create it. Then when you read the data in from the file, simply look for your special symbol. If you are inputting string data, you can easily check to see if you have reached the end of the file by checking to see if the length of the string is zero:

```
IF LEN(A$)<>0 THEN GOSUB 3000\REM If Len<>0 keep inputting data
```

A zero-length string in a data record means that nothing is in that data record and that you are at the end of the file.

14.3 OPENING A DATA FILE FOR INPUT: OPEN Keyword and INPUT File Mode

At the time that you create it, you can only write into (not read from) the file opened in OUT file mode. If you try to input data from the file opened in OUT mode, the system will tell you: That channel not open for input. You can, of course, read from that file if you close the file and then re-open it in INPUT mode.

Only when a data file already exists can you read (input) data from it to your BASIC program. Follow the procedure outlined above, except:

1) Your file specification must select a data file that already exists and that already has data in

it. If the extension of your data file is NOT .DT, you must specify the extension in your file specification; if you do not specify one, BASIC assumes a .DT extension.

2) When reading from a file, more than one open file may be on the same disk.

3) Use the INPUT file mode to tell BASIC that you are going to read from the file.

4) Use the INPUT and INP commands to read data from the file (Section 14.4).

EXAMPLE:

```
FILE:4,OPEN,"<2>Prime-Numbers",INPUT
```

When you open the file, BASIC is pointing to the first data record in the file-- you begin reading data from record #1.

For information on the file-handling commands that actually do the data transferring between BASIC and data files, see Section 14.4

14.4 DATA TRANSFER: PRINT, INPUT, INP, OUT Commands

You already know how to ask for input from the user of your program (INPUT and INPUT1). You also know how to display data on the video screen (PRINT and PRINT,). You may not have realized that in doing so you were using file channels all along. When you say INPUT on a program line, BASIC uses the default file channel of Ø (for communicating with the keyboard), and the PRINT statement uses the default file channel of 1 (for communicating with the video screen). Now, instead of communicating with the keyboard and the screen, you can communicate with a disk file. The methods for doing this are very simple. You use the familiar commands INPUT, PRINT, INP, and OUT, but you tell BASIC that you want to talk to a disk data file. You do this by including a colon followed by a legal file channel number after the command. This file channel number is one that you have already assigned to the data file that you want to communicate with. You have done this by previously using a FILE statement with the OPEN keyword.

14.4.1 Writing Data to a Data File: PRINT and OUT

To output data to a disk data file, you must create the file using a FILE statement. That statement includes the OPEN keyword and the OUT file mode. Once you open the file using the FILE statement, you have a file channel assigned to the created data file. From that point on until you close the data file, you output data from your program into that file using the output file-handling commands, PRINT and OUT. BASIC knows that you are outputting data to a disk file because you follow PRINT and OUT with a disk file channel number (4-7).

14.4.1.1 PRINT

The form of the PRINT statement is:

PRINT:n,print list

where n is the file channel number, and the print list is the one or more elements that you wish to write to the data file. The print list may contain expressions, numerical variables, or constants. PRINT outputs the values of expressions and variables to the file.

EXAMPLE:

```
1)      PRINT:4,A+B,7.5,C1
2)      PRINT:5,A$,"This is a string"
3)      PRINT:7,SQRT(18),3*SIN(A)
```

Formatting Data

When you use PRINT, BASIC automatically places a carriage return symbol after the print list data that you have written to the file using that PRINT statement. Remember that BASIC groups the information in data files into data records. A data record is the series of characters between two carriage return symbols. Example #1 above forms one data record containing 3 elements: the value of A+B, the number 7.5, and the value of the variable C1. PRINT places a carriage return symbol after those items; the next time something is PRINTed to the file assigned to channel #4, BASIC starts a new data record.

You will remember from your previous experience with PRINT that a comma at the end of the print list will keep PRINT from generating a carriage return. When you are communicating with the video screen this means that the next PRINT statement will display its print list on the same line. When you are communicating with a disk file, a comma at the end of a print list means that PRINT will not put a carriage return symbol after that print list (and so will not end that data record). The next print list placed in the file will continue in that data record.

BASIC groups your data into data records so that it can tell where it is within the data file: it keeps track of its position by noting the data record number. How you group your data will become important later when you use the keywords POS and REW to move about within the data file to a particular data record. If you are going to be writing string and numerical data within the same file, See Section 14.4.2, INPUT, for advice on arranging your data in data records.

You can format the data in your print lists by including the BASIC TAB(x) command, where x is an expression, variable, or constant evaluating to a positive integer. This value tells BASIC how many character spaces to move over before printing the next item in the print list. The TAB command allows you to print your data in a file in tabular form.

EXAMPLE:

```
PRINT:5,TAB(INT(I/2)),32,TAB(J),5.23
PRINT:6,TAB(K*5),A$
```

You can also format the numerical data in your data file using the BASIC format strings. The I-format, for example, prints integer numbers right-justified in a field of a specified width. (See Sample Program 14.1 for an example of its use.)

EXAMPLE:

```
PRINT:4,%10I,A,B,C
```

14.4.1.2 OUT

The OUT command transfers one byte or a string of bytes into a data file. One byte is eight bits of data, or one character. The OUT statement takes the form:

```
OUT n,element list
```

where n is the file channel number assigned to the open data file you want to write into. Separate all items in the element list with plus symbols.

EXAMPLE:

```
OUT 5,"a string"+CHR$(13)+"and a carriage return"
```

OUT 0: An OUT 0 will place characters into the keyboard buffer. The keyboard buffer is the area of memory that the system uses to store input from the keyboard. The system has no way of knowing whether the characters in the keyboard buffer were typed in by you or whether you used OUT 0 to put them there. The buffer can hold 64 characters. Once the buffer is full, BASIC will not put more characters into it. Be careful when putting characters into the keyboard buffer; it may already have characters in it that were inputted from the keyboard, so you might not have room for the full 64 characters! An example of the use of OUT 0:

```
100 IF N$="NO" THEN OUT 0,"BYE"+CHR$(13)\ STOP
```

The above example places the BASIC exit command BYE into the keyboard buffer followed by a carriage return. When the program reaches line 100, it evaluates N$. If N$="NO", the program performs the OUT command and then stops. BASIC then looks in the keyboard buffer. In response to the BYE command, BASIC returns the user to the system Exec.

14.4.2 Reading Data From a File: INPUT and INP

If a file already exists and has data in it, you can read that data from it. Open the file in INPUT file mode (Section 14.3). Then you can use INPUT and INP to read data from it.

14.4.2.1 INPUT

Now instead of getting a value or a string from the user of your program, you can input these things from a data file. The INPUT statement takes the form:

```
INPUT:n,input list
```

where n is the file channel assigned to the file you have opened to input. The input list may consist of numerical or string variables.

EXAMPLE:

```
INPUT:5,A,B,C
```

The above statement will input three values from the data file assigned to file channel #5. If you try to input more items than are actually in the data file, you will receive an error message: Input error. If you try to input more things than are in the data record, you will input data from the next data record.

Inputting string data is a little different. INPUT knows where one numerical value ends and the other begins because PRINT places a space between numerical values. When you say:

```
INPUT:5,A,B,C
```

BASIC looks for three numbers in the data file (that is, three numerical items separated by spaces). However, BASIC has no way of separating strings except by carriage return symbols. (After all, since spaces can be a part of strings, there is no way of telling whether a space is part of a string or is separating strings.) Therefore, when you say:

```
INPUT:5,A$
```

BASIC reads from the front of the current data record until it finds a carriage return-- that data is assigned to the value A$.

When BASIC stores data in a data file, it stores the numerical data 3 in the same form as the string data "3". Therefore, when you read data in from a data file, BASIC can't tell if numbers are numerical data or string data. Let's say that the first data record in your file is: 1 2 3. If you say:

```
INPUT:5,A
PRINT A
```

You will get the answer 1. If, however, you had said:

```
INPUT:5,A$
PRINT A$
```

you would have got the answer 1 2 3.

Note: After every INPUT in your program, BASIC points to the next data record:

```
FILE:6,OPEN,"<2>NEW-DATA",INPUT
INPUT:6,A,B,C
INPUT:6,A,B,C
INPUT:6,A,B,C
```

will input the first three values from the first three data records in the file. Make sure that there are actually three numbers in each data record; otherwise you will probably be trying to input the wrong number of items and may run out. This would cause an input error message.

14.4.2.2 INP

The INP command transfers one byte (one character) from the data file. This byte will be in the coded form in which data is stored within the system; that is, in ASCII code. (See Appendix D, Character Set.) Let's say that the first data record in your data file consists of a 1. If you INP the first three bytes of the data file, you will get the following: a 32, a 49, and a 13. The 32 is the decimal ASCII code for a space (the PRINT statement always separates numerical values with a space so that INPUT can tell where one number begins and another leaves off). The 49 is the decimal ASCII code for a 1, and the 13 is the decimal ASCII code for a carriage return symbol (the end of a data record).

The form of the INP statment is:

```
INP(n)
```

where n is the file channel assigned to the data file you have opened for input. If you wish to see the byte displayed, use the command:

```
PRINT INP(n)
```

If you use the INP command but there is no more data in the file (that is, you have reached the end of the file), BASIC will give you back a zero, and will continue to do so from then on; it will not let you go beyond the end of the file.

14.5 CLOSING A DATA FILE: CLOSE keywords and BYE and EXEC

After you have completely finished writing data to a file, you must "close" that file. Until you close a file, the data that you send to a data file may not be physically in the file; it could still be in the data-transfer path to the file, the file channel. To close a file, type:

```
FILE:n,CLOSE
```

where n is the file channel assigned to the file opened in OUT or INOUT mode.

You already know from looking at the System 88 User's Manual that to leave BASIC and return to the system you use the BYE or EXEC commands. When you use the EXEC command, your data files are NOT closed; this is because you can reenter BASIC from the system. When you use the BYE command, BASIC closes all of your file channels, since having used BYE, you cannot reenter BASIC and continue writing to those files.

The BASIC commands RUN, SCR, SAVE, and LOAD all force BASIC to close any open file channels.

If you open a new file on a file channel already in use by another file, BASIC will close the old file to allow the new file to use the channel.

14.6 SELECTING A PARTICULAR DATA RECORD: POS and REW Keywords

Remember that every time you read from a data file, BASIC advances within the file. The next time you read data you will not read the same data you read the last time. A new INPUT statement will advance to the next record. It is often necessary to direct BASIC to a particular data record, or to the beginning of the data file. You will use the POS and REW keywords in a FILE statement to tell BASIC which data record to read from next. You can ONLY use REW and POS on files opened in INPUT or INOUT mode. If you try to use REW and POS on files which you are creating, you will receive the error message: I can't do that to an OUT file.

14.6.1 Rewinding a Data File (REW)

The REW keyword is used to "rewind" a data file; that is, to tell BASIC to begin reading from the first data record in the file. The statement takes the form:

```
FILE:n,REW
```

where n is the file channel assigned to the data file you are reading from. The next time your program reads data from the file, it will read from the first data record.

14.6.2 Positioning a Read to a Particular Data Record (POS)

Use the POS keyword to position BASIC to a particular data record. Remember that the first data record in the file is record #1, the second is record #2, and so on. The POS statement takes the form:

```
FILE:n,POS,record #
```

where n is the file channel assigned to the data file you have opened in INPUT mode. The record # can be any variable, expression or constant evaluating to a positive number less than or equal to 65535.

EXAMPLE:

```
FILE:6,POS,5*5
```

After the statement above is read by BASIC, the next time your program reads data from the input file assigned to channel #6, BASIC will begin reading from data record #25. If you give POS an invalid record number, the next time you try to input data BASIC will display an input error message.

You may position BASIC to any data record within your file, regardless of your present position in the file. This means that you can go forward or backward in the file.

The statement:

```
FILE:6,POS,1
```

is the same as a statement using the REW keyword; both statements will direct BASIC to begin reading at the first data record.

14.6.3 Fast Read Positioning (Fast POS)

Sometimes the data records in a data file will be all of the same length, and sometimes they will be of different lengths. A data file whose data records are of different lengths is called a variable-length record file. When it advances through this kind of file, BASIC finds the end of a data record by looking for a carriage return symbol.

The other kind of data file is called a fixed-length record file. You do not have to tell BASIC that a file is a fixed-length record file; BASIC can figure it out by itself.

[You can see for yourself whether BASIC thinks that a data file has fixed length records. While in enabled mode (See the System 88 User's Manual for an explanation of enabled and disabled mode), list the directory of the disk containing the data file. All data files have load and start addresses of zero except for fixed-length files; they have a non-zero start address. This start address is where BASIC stores the number representing the length of the file's fixed-length data records.]

BASIC treats a fixed-length record file a little differently than it does a variable-length record file. Ordinarily BASIC has no idea how long data records are. Using the POS command on a variable-length record file causes BASIC to search through each data record looking for the start of the next record. If you use POS on a fixed-length record file, however, BASIC now knows exactly how far to advance to reach the next record. Instead of searching through all of the previous data records, BASIC simply calculates the proper number of record lengths to jump to reach the record that you want.

To make sure that all of your data records are the same length, you can use the format strings for numerical data, and for string data you can check the length of the strings, possibly "padding" them with spaces.

14.7 UPDATING DATA RECORDS: INOUT File Mode

We have mentioned before that you cannot read data from a file you have opened in OUT mode. Neither can you write data into a file you have opened in INPUT mode. In only one case can you both read from and write into a file at the same time; that is, after you have opened a file in INOUT mode.

If you open a file in INOUT mode, you can read from that file, change the data, and write it back out again to the file. You can only open an EXISTING file in INOUT mode. After you open the file in INOUT mode, you can use POS and REW to selectively read and re-write data records.

You use the familiar INPUT and PRINT statements to transfer data between your program and the INOUT file. When you write a data record back into a file, the record must be equal or shorter in length to the original data record in that position in the file. If you try to make the new record longer than the original one, BASIC will not place the extra characters in the file; they will be ignored. Nor can you add additional data records to the file. If your new record is smaller than the original, BASIC will simply pad the rest of the record with binary zeroes, characters invisible to the INPUT statements.

BASIC maintains two different "pointers" (think of them as bookmarkers) into an INOUT data file: one for reading data (INPUT) and one for writing data (PRINT). Reading information changes the input pointer; writing data changes the output pointer. Only the POS and REW statements change BOTH of the pointers. Every time you INPUT data you advance the input pointer so that the next read will take in data from the beginning of the next data record. In the same way, every time you PRINT data, you cause the next write operation to begin a new data record.

When you are using a file in INOUT mode you may want to read a record, and then write over that record. You can do so a record at a time, beginning with record #1, by executing pairs of INPUTS and PRINTS. This will read in data and then write over that data. You can also use POS and REW to position yourself exactly in the file before you read and write data. This has an advantage in that it keeps both input and output pointers exactly "in sync" with one another. Since both pointers are maintained separately, you can see that if you do five INPUTS from the beginning of the file, and then do a PRINT, the data will go, not in the sixth data record, but in the first. However, if you use POS to get to the sixth data record, both input and output pointers are set to record #6.

To use a file in INOUT mode, do the following:

1) Open an existing file in INOUT mode:

```
FILE:5,OPEN,"<2>Invoices",INOUT
```

2) Position to the record you want to change (or, if you want to change every record, skip to step #3). Use the POS command:

```
FILE:5,POS,64
```

3) If you want to read the data you are going to change, use an INPUT command:

```
INPUT:5,A$
```

   If you don't need to see the data, skip to step #4.

4) Rewrite the data by using the PRINT command:

```
PRINT:5,B$
```

5) Both input and output pointers are now pointing to the next data record in the file if you have done both an INPUT and a PRINT. You may now position to another data file (POS) or rewind the file to its beginning (REW).

6) When you've finished with the file, close it using the CLOSE statement (Section 14.5).

<u>Important Note:</u> Be very careful when you input and print data!

You MUST be sure that you know what kind of data is in a data record, and how much. For example: let's say that you have three numbers in a data record. You rewrite that record so that it contains only one number. What happens if you position to the front of that record and INPUT A, B, and C? You will get the number in the first data record, but you will also get the first two numbers in the second data record (if there are two numbers in that record, that is). The next time you INPUT you will move to the next data record, and you will skip the last number in the second record.

Make sure that your inputs match the type and amount of data that is in the data file. Note that you cannot write data that is longer than the data in the original data record; BASIC will simply cut down the new data to fit the old record. The way that BASIC determines the length of a record is not by how many items are in it, but by the total length of those items. A data record containing three one-digit numbers is NOT the same length as a data record containing three two-digit numbers. To allow room for later updating, make sure that your original data records are large enough. You can do this by formatting the records properly. (See Sample Program 14.1 below.) You can also "pad" original string data with string blanks, " ".

To make these ideas clearer, let's look at Sample Program 14.1. It demonstrates how to use a file in INOUT mode. When we refer to a particular program line in this discussion, we'll give the line number, surrounded with square brackets [].

The program sets up a data file F$ [40] containing M records [60]. It uses the format string %#5I [110, 290, 410], which says print the following integers right-justified in a field five spaces wide. This makes sure that enough room is left in each data record for later updating.

We then open the file in INOUT mode and update records, either randomly or by request [150]. After the POS statements [270, 390], both input and output pointers are pointing at the same data record, R. We update the record [290, 410]. Note that when we update a record we increment the update counter, B.

The contents of each data record are: 1) the data record number; 2) the number of times you have updated the record; and, 3) a string giving the name of the data file.

```
REM  *              Sample Program 14.1                          *
REM  *  ----------------------------------------------------     *
REM  *  Demonstrate Use of INOUT Mode and POS Keyword            *
REM  *           to Update Data Records                          *
REM  *  -----------------------------------------------------    *
REM
10 DIM F$(1:50),A$(1:60)\REM * F$ is data file name
REM
20 PRINT "Demonstrate updating records in a file opened in "
30 PRINT "INOUT mode, using PCS.  Give me the name of the "
40 INPUT "file you want me to build: ",F$
REM
50 FILE:5,OPEN,F$,OUT
60 PRINT\INPUT "How many records in the file (10-500)? ",M
70 IF (M<10) OR (M>500) THEN 60
REM
REM  *            ---   Build the Data file ---                  *
REM  * Note use of format string (%#5I) to set up data           *
REM  * records; allows room for later updates.                   *
REM
90 PRINT\PRINT "                         ....Writing file ",F$
100 FOR X=1 TO M
REM  * Write record # (and update counter #) for each record *
110 PRINT:5,%#5I,X,0," file ",F$
120 NEXT
130 FILE:5,CLOSE
140 ON ESCAPE GOTO 150
150 PRINT\PRINT "Enter 1 for updating at random, 2 for record "
160 INPUT "select, and 3 to exit the program: ",K
REM  *          --- Open the File INOUT ---                      *
170 FILE:5,OPEN,F$,INOUT\ON ESCAPE GOTO 140
180 IF (K<1) OR (K>3) THEN 140
```

```
190 ON K GOTO 200,330,440
200 REM  * Random updates
210 PRINT\PRINT "50 updates selected at random.  Enter 1 to see each
220 INPUT "record as it is updated; otherwise enter zero: ",F
230 PRINT
240 IF (F<0) OR (F>1) THEN 210
250 REM  * Select records at random, and update them.
REM
REM  * Format String (%#I) keeps data same length as        *
REM  * original data record                                 *
REM
260 FOR I=1 TO 50
270 R=RND(M)\FILE:5,POS,R\ REM * Choose random record #
280 INPUT:5,A,B,A$
290 PRINT:5,%#5I,A,B+1," "+A$
300 IF F=1 THEN PRINT R,":",%#5I,A,B+1," "+A$
310 NEXT
320 PRINT\PRINT "       50 updates completed...."\GOTO 140
330 REM  * User-selected updates.
340 PRINT\PRINT "Select record number between 1 and ",M
350 PRINT "or input zero to stop."
360 PRINT\INPUT "       Record number: ",R
370 IF R=0 THEN 140
380 IF (R<1) OR (R>M) THEN 340
390 FILE:5,POS,R
400 INPUT:5,A,B,A$
410 PRINT:5,%#5I,A,B+1," "+A$
420 PRINT\PRINT R,":",%#5I,A,B+1," "+A$
430 GOTO 360
440 REM  * Exit the program.
450 FILE:5,CLOSE
460 PRINT "Bye..."\OUT 0,"BYE"+CHR$(13)
RUN
```

Section 15

CONNECTING BASIC TO A PRINTER OR SPECIAL DEVICES

Most of Part II discusses how you can transfer data between a BASIC program and a data file on a disk. However, the system looks upon a disk as just another device hooked up to its file channels. We have already mentioned that you can use OUT Ø to communicate with the keyboard buffer. You can also use INP(Ø) to input a byte from the keyboard buffer. In this section we will talk about how you can assign a file channel to a printer and special devices of your own.

15.1 SENDING DATA TO THE SYSTEM PRINTER

The System 88 User's Manual gives you a full explanation of the System Printer Driver. Using that information you can connect your printer to your System 88. The following discussion assumes that you have already "taught" the printer driver about your printer, and that you have "loaded" your printer. That just means that the system is hooked up to your printer and knows how to send data to it.

The first step in using the printer from inside BASIC is to assign the printer a file channel number. Use this statement:

```
FILE:n,LIST
```

(where n is the file channel number) either as a direct or as a program statement. You may use any file channel from 2 through 7. Make sure that you aren't assigning that channel to a data file if you want to use the printer at the same time as you are reading from or writing to files. The following statements all connect the printer to file channel #3:

```
FILE:3,LIST

FILE:3,1+1+1

Q(P)=3\FILE:Q(P),LIST
```

Now we can send data to the printer over the channel we have assigned to it. To send data to the printer, use PRINT:n or OUT n. These may be direct statements or program statements.

EXAMPLE:

```
100 REM Demonstrate use of Printer
110 FILE:3,LIST
120 PRINT:SQRT(9),"Integers and their squares"
130 FOR I=1 TO 10
140 PRINT:3,I,TAB(8),I^2
150 NEXT
```

Use the PRINT statement to send numbers or strings to the printer; use the OUT statement to send single-byte quantities or a string of one-byte values.

Besides sending data from the keyboard or a data file, you can also use the printer in developing, debugging, and documenting programs:

>FILE:3,LIST	Assigns channel #3 to printer
>LIST:3	Lists program to printer
>DUMP:3	Dumps values of variables in your program to the printer
>XREF:3	Cross-references the locations of the variables in your program. Sends this to printer.

15.2 USING SPECIAL DEVICES (DEF)

We mentioned briefly in Section 14 that a special keyword in the FILE statement allows you to connect your own special devices to BASIC. This is the DEF keyword. The kinds of special device that you might connect to BASIC might include a special disk drive, a modem (a device for communicating over phone lines), etc.

Use the DEF keyword ONLY if you are an experienced machine language programmer adept at writing your own machine-language file-management routines. The following paragraphs are intended for those programmers. If you are interested in learning what is meant by terms such as "accumulator," "register," etc., consult one of the many books on the market on 8080 assembly language programming.

The form of a FILE statement containing the DEF keyword is:

```
FILE:n,DEF,address1,address2,address3
```

where n is the file channel you wish to assign to your device.

The three addresses (in decimal) following the DEF keyword refer to three different memory locations. These memory locations are the starting addresses of three different machine language programs that you have written and loaded into memory. These three programs perform the following functions:

Address1: GETCHAR: Input a character from the channel presently in use. You enter the program with the channel number in the Accumulator. You will return from the program with the input character in the Accumulator, with the Zero flag set if the character is a binary zero, and with the Carry flag set if you are at the end of the file. Do not change the contents of any register other than the Accumulator.

Address2: PUTCHAR: Output a character to the channel in use. Enter the program with the character in the B register and the channel number in the Accumulator. Do not change the contents of any registers other than B and the Accumulator.

Address3: CLOSE: Close the device. Enter the program with the channel number in the Accumulator. Do not change the contents of any other register.

For more information on interfacing machine language routines to your system, see Appendix F of this manual, and see the System Programmer's Guide.

Section 16

SAMPLE PROGRAMS AND SUMMARY OF BASIC FILE-HANDLING COMMANDS

16.0

Now that we have discussed the various ways to connect BASIC to the disk system, you may want to see how these commands are actually used in building and updating files. The sample programs below are extremely simple and not very useful in themselves. They will, however, demonstrate how the commands are used in combination with one another. Try typing in these short programs. Once you see them working you will have a clearer idea of how these commands work.

16.1 SAMPLE PROGRAMS

16.1.1 Building a Small Data File with Fixed-Length Records

This program builds a small data file consisting of names and I.D. numbers. If you display the file you will see the contents in this form:

```
ROGERS***
54676543*
SMITH****
67893213*
EDWARDS**
45629482*
```

To allow for fast positioning later we have constructed this file so that the data records are of fixed length. (See next page.)

Sample Program #1

```
10 REM Demonstrates building of fixed-length record file
15 PRINT "Pick a 1 or 2 digit number as a suffix to your",
20 INPUT " data file name: ",F$
25 DIM N$(1:100),W$(1:100),F$(1:2)
30 FILE:5,OPEN,"<3>DATA"+F$,OUT
35 PRINT "Enter END when done."
40 PRINT "Keep your entries to below 11 characters."
45 INPUT "Employee's Name?: ",N$
50 IF N$="END" THEN GOTO 80
55 GOSUB 200
60 INPUT "I.D Number?: ",N$
65 IF N$="END" THEN GOTO 80
70 GOSUB 200
75 GOTO 45
80 FILE:5,CLOSE
85 PRINT "      Finished....."
90 REM Exit program
100 STOP
REM             Gosub prints fixed length data records
200 REM Pad strings out to length of 10 for later fast POS
REM
210 IF LEN(N$)<10 THEN N$=N$+"*"\GOTO 210
220 PRINT:5,N$
230 RETURN
```

16.1.2 Opening a Fixed-Length Record file in INOUT mode

This program takes advantage of the fast-positioning feature of BASIC. The file we constructed above is a fixed-length record file. BASIC positions to each record within that file extremely quickly. Once we update that file it may not still be a fixed-length file.

Sample Program #2

```
10 REM Demonstrates Fast Postioning with Fixed-length records
20 DIM F$(1:2),A$(1:20),N$(1:20)
30 PRINT "Which data file would you like to update?"
40 INPUT "    --DATA",F$
50 FILE:6,OPEN,"<3>DATA"+F$,INOUT
60 PRINT "Which data record do you want to update?"
70 INPUT "(Enter a zero if you're finished): ",N
80 IF N=0 THEN GOTO 190
90 FILE:6,POS,N
100 INPUT:6,A$
110 IF LEN(A$)<>0 THEN GOTO 140
120 PRINT "That data record doesn't exist...Try again."
130 GOTO 60
140 PRINT "Data record contents: ",A$
150 PRINT "Warning: Don't make new record bigger than old one!"
160 INPUT "New data record: ",N$
170 PRINT:6,N$
180 GOTO 60
190 PRINT "We're done...."
200 STOP
```

16.1.3 Updating a File Without Using POS

Yet another example of the use of INOUT files.

Sample Program #3

```
100 REM Demonstrates INOUT mode
110 I=1 \REM record counter
120 DIM F$(1:2),A$(1:20)
130 DIM N$(1:20)
140 PRINT "Which data file do you want to update?"
150 INPUT "--DATA",F$
160 FILE:4,OPEN,"<3>DATA"+F$,INOUT
170 PRINT "Record #",I," : ",
180 INPUT:4,A$\IF LEN(A$)=0 THEN GOTO 260
190 I=I+1\ REM update record counter
200 PRINT A$
210 INPUT "Want to change it (1 for yes, 0 if not)?: ",N
220 IF N=0 THEN PRINT:4,A$\GOTO 170
230 INPUT "O.K. New record?: ",N$
240 PRINT:4,N$
250 GOTO 170
260 PRINT "Done....No more data."
270 FILE:4,CLOSE
280 STOP
```

16.1.4 Outputting Calculations to a Data File

This program does not ask for input from the keyboard.

Sample Program #4

```
10 REM Calculates first 25 powers of 16
20 FILE:5,OPEN,"<2>POWERS",OUT
30 PRINT:5,"The first 25 powers of 16 are:"
40 X=1\PRINT:5,TAB(20),1
50 FOR I=1 TO 25
60 X=X*16
70 PRINT:5,TAB(20),X
80 NEXT
90 PRINT:5."End of file."\FILE:5,CLOSE\STOP
```

16.2 SUMMARY OF BASIC FILE-HANDLING COMMANDS

The following is a list of all of the BASIC file-handling commands available to you.

NOTE: In the command syntaxes given below, the symbol n refers to the number of the information channel to which the data file is assigned. When we speak of "syntax" we are referring to the proper, acceptable form of a statement.

KEYWORDS

CLOSE (see page 90)
Close a data file. Command syntax: FILE:n,CLOSE

DEF (see page 98)
Use the user-written machine language programs designated in the DEF command for file-handling routines. Three hexadecimal addresses are given. Command syntax: FILE:n,DEF,addrs1,addrs2,addrs3

LIST (see page 97)
Assign printer to a file channel. Syntax: FILE:n,LIST

OPEN (see page 84)
Open a data file. May be used to create a data file. Command syntax: FILE:n,OPEN,file mode

POS (see page 91)
Position the reading of data from a data file to a given data record number. Command syntax: FILE:n,POS,record number

REW (see page 91)
Position a file read operation to data record #0. Command syntax: FILE:n,REW

FILE MODES

INOUT (see page 93)
Read data records in from the data file, allow them to

be updated, and write the, back out to the file.
Command syntax: FILE:n,OPEN,INOUT

INPUT (see page 85)
Read data records in from the data file.
Command syntax: FILE:n,OPEN,INPUT

OUT (see page 84)
Write data records into the data file.
Command syntax: FILE:n,OPEN,OUT

DATA TRANSFER

DUMP:n (see page 98)
Send variable values to a pinter or file (n=channel #).

INP(n) (see page 90)
Input one byte of data (one character) from the data file. Command syntax: INP(n)

INPUT:n (see page 89)
Input one data record (one line of characters between carriage returns) from the data file. Command syntax:
INPUT:n,string and/or numerical variable(s)

OUT n (see page 88)
Write one byte of data (one character) or a string of bytes out to a data file. Command syntax:
OUT n,string or numerical variable(s)+string+expression...

LIST:n (see page 98)
List program to printer

PRINT:n (see page 87)
Write one data record (one line of characters between carriage returns) out to a data file. Command syntax:
PRINT:n,string or numerical variable(s),expressions, strings...

XREF:n (see page 98)
Send variable cross-reference to a printer or a data file.

EXIT COMMANDS

BYE (see page 91)
Exit BASIC and return to the system level (Exec). All data files are closed.

EXEC (see page 91)
Recoverable exit from BASIC-- after communicating with Exec, you may resume your operations in BASIC by typing the command CON after a system prompt. Data files are not closed.

Appendix A

THE BASIC ERROR MESSAGES

The System 88 BASIC error messages were designed to be clear and to help in suggesting solutions to problems that may occur when you run a program. If BASIC finds an error in a direct statement, it will refuse to perform that statement and instead will respond with an appropriate error message.

Example:

```
               >
    enter      >FOR I=1 TO 3
    output     I can't do that directly.
```

If an error occurs within a program line, BASIC will find the error when you attempt to run the program. In this case, BASIC prints the program line that contains the error. Underneath that line, BASIC prints the appropriate error message with an arrow pointing at or near the part of the line that is in error.

Example:

```
    enter    >10 Y=3*(SQRT(16)+YCLEPT)
                               ^
             Syntax error
             >
```

All of the BASIC error messages are listed below, with suggestions about what you can do to remedy the situation that caused the error message to appear.

We also list the BASIC error code next to each error message. This is the number which BASIC associates with the message. You will want to know these codes if you use the ERR special variable to check for particular errors. (See Section 11, Debugging BASIC Programs.)

1.1 ERROR MESSAGES

Error code 1045:
    Arg mismatch error

The number of arguments in user-defined function definition is not equal to the number of arguments listed in function call.

Example:

```
    enter    >10 DEF FNX(X)=X/100
             >20 PRINT FNX(1,2,3)
             >RUN

    output   20 PRINT FNX(1,2,3)
                           ^
             Arg mismatch error
```

```
        >
```

Error code 1027:
        Bad argument error

This error may occur if a parameter given to the PLOT function is out of bounds (for example, if X>127 or Y>47). Check to see that your values are within the limits accepted by the function you are using.

Error code 1050:
        Can't continue!

You have asked BASIC to continue execution of a program, but it can't. Perhaps there is no longer a program there, or you have already reached the end of the program. You will use the CON command if you have interrupted the program with a Control-Y or a STOP statement. If you can continue program execution, you will see a double prompt >>; if you cannot, you will see a single prompt >.

Error code 1279:
        Can't do that to an OUT file

You have tried to do something illegal to a BASIC data file which your program has opened in OUT mode. Perhaps you have tried to use a POS command on the file; that won't work because you cannot position a file that is in the process of being created. If you want to input from an OUT file, close the file and reopen it in INPUT mode.

Error code 1058:
        CHAIN programs must be saved with SAVEF

When you try to CHAIN a program into memory (see Appendix B), you must make sure that that program is in token form; that is, that it has been saved with SAVEF. Otherwise, BASIC will not allow you to CHAIN the program in.

Error code 1028:
        Dimension error

You have tried to dimension an array or string array twice within one program. Or you have tried to dimension an array as a direct statement, but in the Run-Time-Environment that array has already been dimensioned. You can also see this error message if you include a variable as an argument in a DIM statement: e.g., DIM X(A).

Error code 1036:
        Division by zero

You tried to divide a variable, expression, or number by zero. BASIC doesn't know how to do that. If you are not sure whether or not you will be dividing by zero, check for that possibility before dividing. For example:

```
55 IF A=0 THEN PRINT "Oops!"\STOP
60 PRINT "Answer is:  ",B/A
```

Error code 1032:
Format error

The usual cause is that you have tried to print data in an incorrect format. For example, BASIC can't print a number in F-format in a field of greater than 25. If you try to do so, you'll get a format error message. In almost every case, a format error occurs because of an incorrect format string.

Error code 1034:
FOR-NEXT error

If you do not next FOR-NEXT loops correctly, you are likely to see this error message. Make sure that the values for your loop variable, index, and step values are correct. Make sure that your loop variable is the same as the variable you specify in your NEXT statement.

Error code 1029:
Function definition error

You called a user-defined function, but you never defined that function-- so BASIC can't find it! Look at your program again, and make sure that you have your function name right.

Error code 1060:
I can only do that to a disk file

You will see this message if you try to use a file-handling command on a non-disk device hooked up to a file channel. For example, if you assign file channel #6 to a printer, and then say: FILE:6,REW, BASIC knows that it cannot rewind a printer.

Error code 1044:
I can't do that directly

You tried to do something directly that BASIC can't do directly. For example, if you say: GOSUB 100 outside of a program, you will get this error message. You can do many things directly. For example, after you run a program, you can change some values of the program variables directly.

Example:

```
>10 A=5
>20 PRINT A^2
>RUN

 25
>A=6
>RUN 20
 36
```

Error code 1033:
I can't find that line

You tried to perform some function on a nonexistent program line. For example, if you try to delete a line that doesn't exist, or try to list a line that doesn't exist, BASIC will give you this error message.

Error code 1042:
Input error

When you write an INPUT statement (either for input from a data file or from the keyboard), you specify whether you want a string or a number. If the item you receive back is not of the correct type, you'll receive the Input error message. The reason that you get an Input error message when you try to input from an empty data file data record is that BASIC tries to input binary zeroes, which it does not recognize as either string or numeric data.

Example:

```
>10 INPUT "Give me a number: ",A
>RUN

Give me a number: HELLO THERE
Input error
```

Error code 1088:
....LOAD interrupted

You interrupted the LOAD command while it was loading a BASIC program. Try again.

Error code 1038:
Missing matching NEXT

There are not enough NEXT statements in your program to match the FOR statements. For example:

```
10 FOR I=1 TO 10
20 FOR J=3 TO 100
30 PRINT I,J
40 NEXT J
50 STOP
```

will not work because there is no NEXT I. Check your FOR and NEXT statements to see if they agree.

Error code 1052:
Nothing to save!

You tried to save a BASIC program, but BASIC decided that there was no program in memory. Try to use the LIST command to see if there is really nothing to save. If you do not see a program listed after using LIST, then you're out of luck. Remember next time to save any program in memory before you leave BASIC or do anything which might endanger the contents of memory.

Error codes 1040,1041:
Oops...BASIC goofed!

You should never see this message; it occurs only if some pointers and parameters inside BASIC become scrambled. If you do get this message, you might want to save your program (to keep it from harm) and try to do again whatever you were doing.

Error code 1047:
Overflow error

You tried to evaluate an expression with a value too large for BASIC to represent. For example: PRINT 3*10^68

Error code 1043:
Out of memory

You managed to fill memory. Revise your program to be more frugal of memory space. Look at Section 10, Optimizing your BASIC Program, for advice on saving memory space. An example of the kind of thing that can fill memory is the following endless loop:

```
10 GOSUB 10
```

Error code 1030:
Out of bounds error

One possible cause for this error message is using a program line greater than the maximum number of 65536. You might see this error if you try to dimension an array to a size greater than memory will hold: DIM X(5000000000).

Error code 1039:
Read error

You tried to read data from a DATA statement, but BASIC became confused. Perhaps there was not enough data in the DATA statement. Then again, the data may not have been in proper form: strings when they should have been numbers, etc. Look at your DATA and READ statements and see if they agree as to type and amount of data needed.

Error code 1035:
RETURN without GOSUB

You must always end a subroutine with a RETURN statement. If BASIC finds an extra RETURN statement, it does not know which subroutine call to return to. You might see this error if you have allowed your program to "fall into" a subroutine. For example:

```
10 IF A=1 THEN GOSUB 200
20 PRINT A
200 REM Subroutine
210 PRINT A*B
220 RETURN
```

The program above causes the subroutine to be executed if A=1. Then when we return to the subroutine call, line 20 is executed. At that point, however, we "fall into" the subroutine. We will then reach the RETURN without a GOSUB having been executed. To avoid this problem, put a STOP at line 30.

Error code 1026:
Subscript error

You tried to use a non-existent subscript or a subscript larger than allowed by the DIM statement. For example, if you've previously said that A=50, the following will generate a subscript error:

```
10 DIM N(20)\PRINT N(A)
```

Error codes 1024,1025:
Syntax error

Syntax means "arrangement." In the case of BASIC, syntax means the correct form of a command that BASIC can understand.

"Syntax error" is the most common error message that you will see. In general, all of the other error messages occur because BASIC understands what you want (or what you say you want), but can't do it. A syntax error message is BASIC's suggestion to you that your command was not in the proper form: either you misspelled a command or you did not write the command in the correct way. For example, BASIC will respond with a syntax error if you say:

```
PRIMPT A              (you meant PRINT)
IF X=0 GOTO 200       (THEN keyword missing)
```

Error code 1053:
That channel not open!

You tried to use a file channel that your program has not previously assigned to a data file or printer. Use the FILE statement (see Section 14) to attach a file or printer to the channel or change the channel number in your program to reflect the file channel you HAVE assigned to those devices.

Error code 1054:
That channel not open for input

You did an INPUT or INP command, but the file channel you specified is not assigned to a file opened in INPUT or INOUT mode. Check your FILE statements to see if you are using the correct file channel number and to see if you used the correct file mode.

Error code 1055:
That channel not open for output

You did an OUT or a PRINT command, but the file channel you specified is not assigned to a file opened in OUT mode. Check your FILE statements to see if you are using the correct file channel number and to see if you used the correct file mode.

Error code 1046:
That line was too long!

BASIC limits the length of a line to 128 characters. If you try to make a line longer than that, you're going to see the above error message. Try to split your program lines up so that the commands appear on separate lines. You can split up an input prompt, for example, by using a PRINT statement.

Example:

```
10 PRINT "This input prompt was too long for one ",
20 INPUT "line, so now it's on two lines: ",N
```

Error code 1057:
That program is for a different version of BASIC!

The program you loaded into BASIC was written in another version of BASIC. It won't run with this BASIC. If it is in non-token, ASCII form, you can edit it using the System 88 text editor so that it will run with this version of BASIC.

Error code 1063:
That's not a BASIC data file

You tried to use a file as a BASIC data file, but BASIC didn't recognize it as a data file. If the extension of the data file is not .DT, you must specify the extension in your FILE statement. Try again, but this time explicitly state the data file's extension.

Error code 1051:
That's not a BASIC file!

You tried to LOAD a non-BASIC file. Make sure that the file you tried to load in is indeed a BASIC program. Make sure that you specified the file's extension if that extension was not the default extension .BS.

Error code 1031:
Type error

You tried to use a string function on a numerical variable or vice versa. For example, PRINT SQRT(A$) or PRINT LEN(N) are both incorrect uses of the numeric and string functions.

Error code 1062:
Type error on READ

You will see this error message if you try to READ string data from a DATA statement containing numerical data or vice versa. Check your DATA and READ statements for type and amount of data.

Appendix B

RUNNING BASIC AND LOADING AND SAVING BASIC PROGRAMS

1. RUNNING BASIC OR BASIC PROGRAMS

To run BASIC on the System 88, just type the command:

```
BASIC
```

after a single or double dollar sign prompt. The Exec then loads BASIC into memory and runs it.

Most BASIC program files carry the extension .BS (automatically appended to the file name by the system); to run a BASIC program with the extension .BS, just type the file name while in Exec. This works only if the extension on the file is .BS. For example, if you have the file Loans.BS on the System Disk, you can run it by typing:

```
Loans
```

after the single or double dollar sign prompt. When the system goes out to get the file Loans, it notes the .BS extension and automatically loads BASIC to run the file.

But if the file Loans was created with some other extension-- if you specified some other extension when first creating the file-- then the system cannot tell that the file is a BASIC program. Instead, you must explicitly tell the system to have BASIC run the file. For example, if you have the file Loans.GG on disk 2, the command:

```
$<2>Loans
```

will generate an error message; the system doesn't know what to do with an extension of .GG on a file. The command

```
$BASIC <2>Loans.GG
```

will cause BASIC to load and run your file.

2. LOADING PROGRAMS FROM BASIC

You can load a file from the disk while in BASIC by using the LOAD command. The following example shows how to load the file <3>Eigenvalues.BS:

```
>LOAD,<3>Eigenvalues
```

(> or >> is the BASIC prompt).

Remember, the extension on the file does not have to be specified if it is .BS, which is the default extension used by BASIC when no extension is specified. If you wanted to run a BASIC program saved as PROGRAM.WW, you would have to type:

```
>LOAD,PROGRAM.WW
```

to explicitly tell BASIC that although the file does not have a .BS extension, it is indeed a BASIC program.

Note that when you invoke a BASIC file from the Exec level, either by typing its name or by typing BASIC followed by the file name, the program you want is loaded into memory and automatically run; you don't have to give BASIC the RUN command when you invoke files this way. If you load files from within BASIC using the LOAD command, you have to give BASIC the RUN command to start the program, unless the program was saved in "auto-execute" mode (described below).

When you load files using the LOAD command in BASIC, any program lines in memory at that time are NOT removed. This provides us with a way of merging programs together; you can develop parts of a larger program in small sections, or load already debugged subroutines to the program in memory. This also means that you must give the SCR command to remove all program lines in memory if you do not want your program merged with the program in memory.

When BASIC is loading a program from disk, some direct commands are allowed in the file. These commands include: REM, PRINT, IF, FILE, SCR, and CLEAR. If BASIC sees a statement it does not understand, it stops the loading process and gives the Syntax error message.

Placing REM statements in the file as direct statements offers the advantage that the comments are available in the disk file but do not take up room in memory when the BASIC program is being run. REM statements are placed in a file this way using the text editor, rather than from within BASIC.

3. SAVING BASIC PROGRAMS

Programs may be saved by using two similar commands: SAVE and SAVEF. Both commands save all the program lines (but not the data) in memory as a disk file. SAVE saves the program on the disk in text form that may be changed using the text editor, and printed using the system TYPE or PRINT commands. (See the User's Manual for more information on the Editor, TYPE, and PRINT.) SAVEF saves the program in BASIC's internal format. This makes programs saved with SAVEF faster to load (sometimes two or three times faster), but these files may not be edited using the text editor or run by other versions of BASIC (such as different arithmetic precision versions). SAVEF must be used to save programs to be used with CHAIN in BASIC, described later. When a SAVEF file is loaded, it automatically begins execution. A program saved with the SAVE command will not start execution automatically unless it was specifically saved in "auto-execute" mode. This is done by separating the file name from the BASIC command SAVE with a semicolon ; rather than a comma , . The following examples show this:

Using SAVEF

```
>SAVEF,Program-1
```

Using SAVE

```
>SAVE,<3>Program-2
```

Using SAVE with auto-execute

```
>SAVE;Program-3
```

Using SAVE and specifying an extension

```
>SAVE,Program-4.XX
```

These examples saved the BASIC program in three different ways. If you don't specify an extension on the file name, BASIC uses .BS, which identifies the file as a BASIC program. The examples above produced files Program-1.BS, Program-2.BS, Program-3.BS, and Program-4.XX. File Program-1 can't be changed using the text editor, and if you TYPE the file, it will not look like a BASIC program. Files Program-2, Program-3, and Program-4 can be TYPEd or PRINTed and can be changed using the text editor. If you examine these files, you will see that the auto-execute feature just puts the BASIC command RUN at the end of the program file.

4. LOADING PROGRAMS SAVED BY POLY 88 BASIC

Programs saved on cassette are loaded onto disk using the FILMS processor described in the User's Manual. Programs written for POLY 88 BASIC will have to be edited before they can be run on the System-88 if they: 1) use strings and string subscripting; 2) alter the video board by using POKE and PEEK; or, 3) use the POLY-88 printer driver. Additionally, computational programs can be speeded up by re-writing them to use the MAT statement.

5. CHAIN

The CHAIN capability in BASIC allows the programmer to break up large programs and have only part of the program in memory at a time. Other parts of the program are loaded by name, using the CHAIN statement in BASIC. The programs loaded by CHAIN must have been saved using SAVEF in BASIC. Two examples of the CHAIN statement are:

```
230 CHAIN "Part-4"
250 CHAIN STR$(K)+"-B"
```

Line 230 loads into memory file "Part-4." If K has the value 10, line 250 will load and run file "10-B." This second example shows that the file name used by CHAIN may be a string expression. The file name could also be an element in a string array.

The important thing to remember in using CHAIN is that the values of all variables and the status of all file channels are preserved, but "control" information, such as FOR-NEXT loops, GOSUB data, etc., is lost. Here is a detailed description of how CHAIN works:

1) The string expression for the file name of the program to CHAIN in is evaluated and copied into a special area of memory. The file specified is looked up. If the file is not found, an error is generated.

2) The file is checked to make sure it is a SAVEF file built by this version of BASIC. An error is generated if it is not.

3) The BASIC control stacks are erased. This means that FOR-NEXT loop information, GOSUBS, and function calls are lost. Practically, this means that CHAIN cannot be done from within a FOR-NEXT loop, a GOSUB-called routine, or from within a function. The values of all scalars and arrays are preserved, as is the status of all the file channels.

4) The first line number in the CHAIN file is extracted and saved. Program lines in the existing program are deleted from this line number to the end of the program.

   This means that if the first line of a CHAIN program is line 1000, lines 1000 through the end of the program in memory will be deleted before the CHAIN program is loaded.

5) The CHAIN program is loaded into memory. When the end of the file is reached, program execution is started at the first line of the CHAIN file, leaving all variables and files intact. If the first line in the CHAIN program was line 500, this would be the same as giving BASIC the command RUN 500.

CHAIN loads in the new file program line by program line. This means that for each line that is inserted, all the variables and file information must be moved around. If you have large arrays and other data, this process may be slow.

6. BASIC PROGRAMS AS SYSTEM FILES

On the System 88, files marked "system files" may not be typed, printed, renamed, or deleted. BASIC programs may be protected by making them system files. BASIC programs marked system files may not be saved or altered. In fact, the only commands that are allowed in BASIC for system files are SCRATCH, BYE, and RUN (without a line number). Commands such as LIST, DUMP, XREF, and the altering of program lines are not allowed. Additionally, BASIC programs cannot access system files unless the extension of the file being accessed is .DT, indicating it is a BASIC data file. If the file with the extension .DT is a system file, the program accessing it must also be a system file. These features allow programmers to protect their programs and data from the casual user. The System Programmer's Guide (available separately) discusses how to make files system files.

Appendix C

SAMPLE PROGRAMS

These programs of varying complexity are listed in this manual so that the user can type them in and see various BASIC features in execution. The programs in this section were contributed either by R. T. Martin, W. W. Hogg, or S. Tytonida.

The user will note that the REM statements in the programs are given without line numbers. These programs were written with the aid of the System 88 Text Editor, which allows one to carefully document BASIC programs without cluttering machine memory with remarks.

The names of the eight sample programs are:

ROSES
ORBIT
PRIMES
RHIST
SORT
CLOCK
TIMER
GRAMMAR

Sample Program ROSES

This program is a "number cruncher." A number cruncher is a program that does an extraordinary amount of computation. For each point displayed on the screen, two sines and a cosine must be calculated (line 190). If 24K or more memory is available, these values for sin(t) and cos(t) may be precomputed and saved in an array, thus eliminating a good portion of the computation. The number of sample points computed is set as variable K on line 100. This number may be increased, increasing the intricacy of the pattern as well as the time required to "draw" each curve. Try values of N larger than 100 (or even 1000) and observe the results. Try K = 500 and starting N = 83. If you are mathematically inclined, examine the effect of sampling the rose equation in closed form. Why is it that for N > 1000 we do not see a solid white screen (for K = 500), but instead see some very interesting patterns?

```
REM             SAMPLE PROGRAM "ROSES"
REM
REM     This program plots roses on the video screen.
REM     The general form of the rose, in polar form, is
REM     R=A*SIN(N*T) where A is the maximal radius, and
REM     T is the angle theta, which goes from Ø to 2*PI
REM     radians to generate the rose.  To plot this function
REM     in the Cartesian coordinate system, we use the
REM     transformations X=R*COS(T)+X1 and Y=R*SIN(T)+Y1,
REM     where (X1,Y1) are the coordinates of the point we
REM     wish to call the origin.  This gives us the equations
REM     X=63.5+44*SIN(N*T)*COS(T), Y=23.5+22*SIN(N*T)*SIN(T)
REM     To speed up the computation, we factor out the term
REM     SIN(N*T) to give the equations shown below.  Note
REM     that we only computer K points along the curve; this
REM     gives us an interesting sampling effect for large N.
REM     We input a starting N, and generate roses for N
REM     decrementing down to 2.
REM
REM
REM     Change K for more or less points.
REM
1ØØ K=1ØØ
11Ø PRINT CHR$(12),"SAMPLE PROGRAM ROSES"
12Ø PRINT"I will plot the equation for a family of roses based"
13Ø PRINT "on the starting number you give me (>2, please!)."
14Ø INPUT "STARTING N = ",L
15Ø IF L<2 THEN PRINT"...GREATER THAN 2, PLEASE!" \ GOTO 14Ø
16Ø FOR N=L TO 2 STEP -1
17Ø PRINT CHR$(12),\ PRINT "N =",N \ PLOT Ø,44,Ø
18Ø FOR T=Ø TO 2*PI STEP 2*PI/K
19Ø S=SIN(N*T) \ X=63.5+44*S*COS(T) \ Y=23.5+22*S*SIN(T)
2ØØ PLOT X,Y,1 \ NEXT
21Ø NEXT \ GOTO 1ØØ
RUN
```

Sample Program ORBIT

The ORBIT program simulates the motion of two massless particles in motion about a force center. Describing them as "massless" particles is another way of saying that they do not interact with one another. They interact only with the force center.

This program was run with a POLY 88 driving an Advent Corporation projection television system, producting an image approximately five feet across, and was quite entertaining.

Try changing the value for D on line 130, which controls the accuracy (step size) of the approximation. Also try altering (slightly at first) the initial conditions for the particles, such as the velocity components set by V1, V2 and V3, V4.

This program was written during a visit to the Physics Computer Development Project (PCDP) at the University of California at Irvine. The idea for the program was suggested by Dr. Richard Ballard, who was interested in seeing what a PolyMorphic Systems microcompuuter would do with another "number cruncher," such as a very simple model of motion in a force field. Dr. Ballard described the functions and they were turned into ORBIT.

ORBIT is dedicated to Isaac Newton, who was able to connect the motion of the planets to an apple falling from a tree.

```
REM              SAMPLE PROGRAM "ORBIT"
REM
REM      Demonstrates plot function in displaying the
REM      orbits of two massless particles about a force center.
REM      Simple 2 body orbital kinematics program.
REM      Kinematics equations by R. Ballard, programming
REM      by R. Martin, basic understanding and explanation
REM      of motion by I. Newton.
REM
REM      NOTE: ORGANIZED FOR SPEED, NOT EXECUTION!!!
REM
100 PRINT CHR$(12), \ PLOT 0,47,0
110 PLOT 50,25,0 \ PRINT CHR$(128+14) \ PLOT 0,21,0
120 X1=3 \ X2,V1,T=0 \ V2=.5 \ D=.1
130 D=.5 \ REM change D for more or less accuracy in orbits
140 X3=2 \X4,V3=0 \ V4=-.6
150 PLOT H,V,0 \ H=10*(X1+5) \ V=5*(X2+5) \ PLOT H,V,1
160 PLOT H1,H2,0 \ H1=10*(X3+5) \ H2=5*(X4+5) \ PLOT H1,H2,1
170 X1=X1+V1*D \ X2=X2+V2*D \ X3=X3+V3*D \ X4=X4+V4*D
180 S=X1*X1+X2*X2 \ R=SQRT(S) \ S=D/(R*S) \ V1=V1-S*X1 \ V2=V2-S*X2
190 S1=X3*X3+X4*X4 \ R1=SQRT(S1) \ S1=D/(R1*S1) \ V3=V3-S1*X3
200 V4=V4-S1*X4 \ T=T+D \ GOTO 150
RUN
```

Sample Program PRIMES

This program was originally written to fill the need for a program that would compute continuously for system testing. It simply computes prime numbers, displaying the last computed number on the screen. In the calculation itself, we keep in vector N; a list of up to the first 500 primes to use as trial divisors in testing a number for being prime. If a number does not have a prime divisor less than or equal to the square root of the number, it is prime. In the calculation we use L as a pointer into the list of prime divisors in a way that alleviates the need to compute the square root for each new number. This technique was described by Ira Baxter to R. T. Martin in a conversation in 1971. Those interested in prime numbers might look at Volumes 1 and 2 of The Art of Computer Programming by Donald E. Knuth, published by Addison-Wesley.

```
REM              SAMPLE PROGRAM "PRIMES"
REM
REM              Find and print prime numbers.
REM              MARCH 1977, S. TYTONIDA
REM
REM      The list N is used to hold the first 500 primes
REM      In testing to see if a number is prime, we only need
REM      to look for factors that are less than or equal to
REM      the number; in fact, we only need to check prime
REM      factors less than or equal to the square root of the
REM      number.  Rather than calculate a square root every time
REM      we instead keep a pointer, L, into the list of past
REM      primes, and bump that up as needed. note that we only
REM      test odd numbers. The number we display in the middle
REM      of the screen is the latest prime, the number at the
REM      bottom is the current test bound.  The rather
REM      baroque expression (INT(M/N(P))*N(P)-M) gives the
REM      remainder of dividing the number M by prime factor N(P).
REM      If the remainder is zero, the number cannot be prime.
REM      If non-zero, we must test prime factors thru N(L).
REM      If none of those are divisors, we have a new prime,
REM      and if K<500, we stuff it onto the list. My thanks
REM      to Ira Baxter for explaining to me, many moons ago,
REM      why you don't need to calculate square roots every
REM      time, and to the ancient Greeks that discovered the
REM      magic and madness of prime numbers.
REM
REM      REMEMBER: (2^19937)-1 IS PRIME!
REM
REM
100 DIM N(500)
110 PRINT CHR$(12),\ PLOT 0,47,0
120 N(1)=2 \ N(2)=3 \ N(3)=5
130 K=2 \ L=2 \ M=5
140 P=1 \ IF M>N(L)^2 THEN L=L+1 \ GOTO 140
150 IF(INT(M/N(P))*N(P)-M)=0 THEN M=M+2 \ GOTO 140
160 IF P=>L THEN 170 ELSE P=P+1 \ GOTO 150
170 K=K+1 \ IF K<500 THEN N(K)=M
180 PLOT 55,23,0 \ PRINT M," IS PRIME!" \ PLOT 0,20,0 \ M=M+2
190 GOTO 140
RUN
```

Sample Program RHIST

This program was written to provide some analysis of the random number generator used in BASIC. It also uses the PLOT feature to produce the histograms and in positioning the cursor for PRINT statements. We compute the distribution of the random number generator cumulatively into 100 "buckets": the array A. We then compute the area under this curve, used in determining the 10% points, and the maximum value in a bucket over the set of buckets, which is used in scaling the histogram bars. This computation is done in lines 150 to 190. We then find the points, or bucket numbers, corresponding to 10% increases in area under the curve.

Note the use of the PLOT statement in line 230 to position the cursor for the PRINT statement producing a carriage return at the end of the line. As an optimization, we do not reprint one of these "decile points" unless it has changed. The remainder of the program is responsible for updating the histogram bars and the scaling of the display. Line 370 computes the scaled height of the histogram bar, and then we will shrink it, grow it, or leave it alone, depending on what is needed. The long-term behavior of a good random (pseudo-random) number generator should produce a relatively flat histogram, and the decile points along the right edge of the screen should be multiples of 10, from 10 to 100.

For more analysis of random number generators, see Volume II of The Art of Computing Programming by Donald E. Knuth; chapter three of this book is devoted entirely to random numbers, psuedo-random numbers, and methods of testing and generating them. The random number generator used in BASIC was provided by Eric Rawson.

```
REM              SAMPLE PROGRAM "RHIST"
REM
REM      Uses the plot function and produces a histogram
REM      showing the distribution of the random number
REM      generator and percentage distributions.
REM
REM
100 DIM A (100),Y(100),Q(10)
110 PRINT CHR$(12), \ PLOT 0,47,0
120 S,N=100
130 FOR I=1 TO 100 \ Y(I)=7 \ NEXT
140 PLOT 121,43,0 \ PRINT "%%%" \ PLOT 0,40,0
150 FOR I=1 TO N \ K=RND(100) \ A(K)=A(K)+1 \ NEXT
REM
REM      H is highest number seen, M is sum.
REM
160 H=-3 \ M=0
REM
REM Compute sum (area under curve) and find high value.
REM
170 FOR I=1 TO N \ M=M+A(I) \ IF A(I)>H THEN H=A(I)
180 NEXT
REM
REM      Put up decile (%%%) points.
REM
190 F=.1 \ G=0 \ J=1
200 FOR I=1 TO N \ G=G+A(I) \ IF G<F*M THEN 240
210 IF Q(J)=I THEN 230
REM
REM      Print point.
REM
220 PLOT 118,3*J+10,0 \ PRINT I \ PLOT 0,3+J+7,0
230 Q(J)=I \ J=J+1 \ F=F+.1
240 NEXT
250 PLOT 0,3,0 \ PRINT "N =",S," MAX =",H \ PLOT 0,0,0
REM
REM      Now plot bars. Note that we scale, so that the
REM      largest bar is 39 high. X=2+I+INT((I-1)/10)
REM      generates a blank spot every 10 to aid in counting
REM      the bars on the screen.
REM      We see if a bar has changed, has grown, or what, and
REM      do the right thing for each case to optimize.
REM
260 FOR I=1 TO 100 \ V=7+INT(39*A(I)/H) \ X=2+I+INT((I-1)/10)
270 IF V=Y(I) THEN 310
280 IF V<Y(I) THEN 300
290 FOR J=Y(I) TO V \ PLOT X,J,1 \ NEXT \ GOTO 310
300 FOR J=Y(I) TO V STEP -1 \ PLOT X,J,0 \ NEXT
310 Y(I)=V \ NEXT
320 S=S+N \ GOTO 150
RUN
```

SAMPLE PROGRAM SORT

Sort was written to demonstrate two different methods of sorting and their relative efficiency. Sort also demonstrates the utility of a microcomputer with the right balance of software features in computer science education. One of the authors (Martin) feels he learned more about sorting algorithmic analysis by sitting down with Vol. III of Knuth and Poly BASIC and building sorting algorithms and testing them than he did in a term of formal classes.

This program also demonstrates the use of PEEK and POKE for examining and modifying memory locations, especially the video card memory, and the use of the TIME function for timing processes.

The interested user is directed to Volume III of The Art of Computer Programming, by Donald Knuth, which is devoted entirely to sorting and searching, rather than Volumes I or II.

```
REM                SAMPLE PROGRAM "SORT"
REM
REM     This program uses the peek and poke functions to
REM     manipulate the contents of the video board, and
REM     more important, demonstrates two techniques of
REM     sorting information: the venerable bubble sort
REM     and the simple but vastly superior "shell" sort.
REM
REM     A good way to randomize..
REM
100 Z=RND(TIME(1)/65536)
REM
REM     Holds stuff to sort
REM
110 DIM P(256)
REM
REM     Holds increments for use by shell sort.
REM
120 DIM H(10)
REM
REM     Calculate increments for shell sort algorithm
REM
130 H=4 \ FOR I=1 TO 10 \ H(I)=H \ H=3*H+1 \ NEXT
REM
REM     Generate list of things to sort.
REM
140 GOSUB 350
REM
150 PRINT CHR$(12),\INPUT "How many things to sort (2-256)?",N
160 IF N>256 OR N<2 THEN 150
170 PRINT "Which sort do you want to use:"
180 PRINT "      1 BUBBLE SORT"
190 PRINT "      2 SHELL SORT"
200 INPUT"1 for BUBBLE, 2 for SHELL : ",M
210 IF M<>1 AND M<>2 THEN 200
220 INPUT "Do you want the same test pattern (Y or N)?",A$
230 IF A$="N" THEN GOSUB 350 \ GOTO 250
240 IF A$<>"Y" THEN 220
REM
REM     This is the screen origin (1800 hex) - 1
REM
250 O=6143
REM
260 PRINT CHR$(12), \ PLOT 0,47,0
270 FOR I=1 TO N \ POKE I+O,P(I) \ NEXT
280 S=TIME(0) \ W=0
290 ON M GOTO 370,430
300 PLOT 0,12,0 \ PRINT "Sorted ",N," things in",W," swaps,",
310 PRINT "and", TIME(1)/60," seconds."
320 INPUT "Try again (Y or N)?",A$ \ IF A$="Y" THEN 150
330 IF A$<>"N" THEN 320
340 STOP \ GOTO 150
REM
REM     Generate new pattern in P
REM
350 PRINT "Thinking...."
360 MAT P=127+RND(127) \ RETURN
```

```
REM
REM     Bubble sort. We wander down the list, looking for
REM     two elements out of order, and swap 'em when we find 'em.
REM
370 S=TIME(0)
380 K=N
390 F=0 \ FOR I=O+1 TO O+K-1
400 L=PEEK(I) \ M=PEEK(I+1) \ IF L<=M THEN 420
410 F=1 \ POKE I+1,L \ POKE I,M \ W=W+1
420 NEXT \ K=K-1 \ IF F=0 THEN 300 ELSE 390
REM
REM     Shell sort. This is from Knuth volume III, algorithm D.
REM
430 S=TIME(0) \ W=0
440 FOR Q=1 TO 9 \ IF H(Q+1)>N THEN EXIT 460
450 NEXT
460 FOR J=Q TO 1 STEP -1
470 F=0 \ H=H(J) \ FOR I=O+1 TO O+N-H
480 L=PEEK(I) \ M=PEEK(I+H) \ IF L<=M THEN 500
490 F=1 \ POKE I,M \ POKE I+H,L \ W=W+1
500 NEXT \ IF F>0 THEN 470
510 NEXT \ GOTO 380
RUN
```

Sample Program CLOCK

This program demonstrates the real-time clock function available in BASIC. It also uses formatted print in displaying the time (lines 190 and 330), PEEK, POKE, and OUT. Without redevelopment, CLOCK turns the System 88 into a very expensive and inaccurate clock. After the program was written, we determined that it loses two or three minutes an hour. Solve the problem of this inaccuracy, and in so doing you will learn about use of the time function. It is also a simple matter to modify the program to display every second.

```
REM             SAMPLE PROGRAM "CLOCK"
REM
REM     This program demonstrates the use of the real time
REM     clock available through the BASIC "TIME" function.
REM     If you have an AI Cybernetics model 1000 speech
REM     synthesizer at output port 254, it will generate
REM     "tick-tock" noises....
REM
REM     Written March 1977 S. Tytonida
REM
100 PRINT CHR$(12),"SAMPLE PROGRAM CLOCK"
110 PRINT "After you give me the current time in hours and"
120 PRINT "minutes, I will be a clock!"
130 INPUT "What hour is it (0-23)",H
140 H=INT(H) \ IF (H<0) OR (H>23) THEN 130
150 INPUT "What minute do I start with (0-59)?",M
160 M=INT(M) \ IF (M<0) OR (M>59) THEN 150
REM
REM     Seconds counter
REM
170 S=0
180 PRINT " When you hit return, I will start being a clock at"
190 PRINT %2I,H,":",M,":",0," o'clock ",
200 INPUT "(hit return to start)",A$
210 PRINT CHR$(12), \ PLOT 0,47,0
REM
REM     "TICK" for a Cybernetics board
REM
220 K=43
REM
REM     The clock symbol
REM
230 W=220
REM
240 O=6144+32+8*64
250 A=TIME(0)
260 IF TIME(1)<60 THEN 260
270 IF K=43 THEN K=47 ELSE K=43
280 IF W=220 THEN W=175 ELSE W=220
290 OUT 254,K \ POKE O,W \ OUT 254,0
300 S=S+1 \ IF S<>60 THEN 330 ELSE S=0
310 M=M+1 \ IF M<>60 THEN 330
320 M=0 \ H=H+1 \ IF H=24 THEN H=0
330 PLOT 0,47,0 \ PRINT %2I,H,":",M,":",S \ PLOT 0,43,0
340 GOTO 250
RUN
```

Sample Program TIMER

This program was included to allow the user to time statements (as described in Section 10 of this manual), to demonstrate the use of the TIME function, and to show that saying NEXT I is indeed slower in resulting program execution than simply saying NEXT. Because even the relatively slow 8080 processor and BASIC can execute statements much faster than 60 ticks per second will allow us to time directly, we must time a known number of these operations and calculate the individual times from that. Any software timing process we can accomplish in BASIC involves the introduction of overhead*, so we must measure that overhead and factor it out of the timings we generate. This is the reason we average over 1000, for the number of operations to time. In the timer program shown, how accurate and repeatable are the results? If averaging over 1000 samples is better than 100, wouldn't one million samples be better? How much better?

The user is especially encouraged to compare the times for various processes when using the MAT statement (see section 8) compared with similar FOR-NEXT loops.

*Overhead time is time taken up doing things other than what we want to do.

```
REM                    SAMPLE PROGRAM TIMER
REM
REM                 (S. TYTONIDA, MARCH 1977)
REM                  (W.W. HOGG, MARCH 1978)
REM
REM     Generate timing information for BASIC programs.
REM     Calculate average timing over 100 samples.
REM
REM     First calculate loop overhead for 100 iterations.
REM
100 T=TIME(0)
110 FOR I=1 TO 100
120 NEXT
130 T=TIME(1) \ T1=T
REM
140 PRINT"Loop overhead is about",T/(100*60)," sec per iteration"
REM
REM     Now time overhead when we use "NEXT I"
REM
200 T=TIME(0)
210 FOR I=1 TO 100
220 NEXT I
230 T=TIME(1)
REM
240 PRINT"versus",T/(100*60)," sec per iteration for NEXT I"
REM
REM Now time A=300
REM
300 T=TIME(0)
310 FOR I=1 TO 100
320 A=300
330 NEXT
340 T=TIME(1)-T1
REM
350 PRINT"A=300 takes about",T/(100*60)," seconds to do."
REM
REM     Now set B=300 and time A=B
REM
400 B=300
410 T=TIME(0)
420 FOR I=1 TO 100
430 A=B
440 NEXT
REM
450 T=TIME(1)-T1
REM
460 PRINT"A=B, for B=300, takes about",T/(100*60)," seconds."
RUN
```

Sample Program GRAMMAR

This program illustrates the use of string arrays. We also make extensive use of the MAT feature of System 88 BASIC. By changing the entries in the DATA statements or altering the grammatical structure that the program uses, you can generate text of your own. Have fun!

```
REM                 SAMPLE PROGRAM GRAMMAR
REM
REM               (W. W. Hogg,  April 1978)
REM
REM
REM     Note that we leave space for blanks, etc. in A$
REM
10 DIM A$(7:12),A2$(4:6)
15 DIM A1$(5:6),N$(20:10),N1$(10:10),V$(20:10)
REM
REM     this is the way we read the data, read the data....
REM
20 MAT READ A1$ \ MAT READ N$ \ MAT READ N1$ \ MAT READ V$
25 MAT READ A2$
REM
REM     Clear out the previous sentence
REM
30 MAT A$=""
REM
REM     and generate the new one.
REM
40 A$(1)=A1$(RND(5))
50 A$(2)=" "+N1$(RND(10))
60 A$(3)=" "+N$(RND(20))
70 A$(4)=" "+V$(RND(20))
80 A$(5)=" "+A2$(RND(4))
90 A$(6)=" "+N1$(RND(10))
100 A$(7)=" "+N$(RND(20))+"."
REM
REM     Slight delay so you can read it..
REM
105 PAUSE 20
REM
REM     print it to the video, and loop.
REM
110 MAT PRINT A$, \ PRINT \ PRINT \ GOTO 30
REM
REM     articles
REM
1000 DATA "The","A","My","The","Frog's"
REM
REM     subjects and objects
REM
1010 DATA "fox","lizard","dragon","unicorn","oyster"
1020 DATA "dog","cat","canary","dodo bird","hyena"
1030 DATA "whale","shark","guppie","snake","virus"
1040 DATA "little boy","old man","triffid","griffin","widget"
REM
REM     adjectives
REM
1050 DATA "green","spotted","tired","hungry","sleepy"
1060 DATA "angry","frightened","happy","righteous","evil"
REM
REM     verbs
REM
1070 DATA "swallowed","devoured","attacked","struck"
1080 DATA "loved","wanted","fed","cleaned"
```

```
1090 DATA "kept","killed","heard","saw"
1100 DATA "found","fought","sheltered","was"
1110 DATA "became","worshiped","inspired","ate"
REM
REM     more articles
REM
2000 DATA "a","the","the","his"
REM
```

Appendix D

THE BASIC CHARACTER SET

All characters and symbols in BASIC are stored in the machine as numbers (the numbers assigned by the ASCII code). The following lists contain all of the characters in BASIC and their ASCII code in decimal representation. To print any character, type PRINT CHR$(number), using the decimal number as given next to the desired character below.

Example:

```
    enter    >LIST
             10 PRINT TAB(10),CHR$(66),CHR$(32),CHR$(65),
             20 PRINT CHR$(32),CHR$(83),CHR$(32),CHR$(73),
             30 PRINT CHR$(32),CHR$(67),CHR$(13),TAB(11),
             40 PRINT CHR$(33),CHR$(32),CHR$(33),CHR$(32),CHR$(33)
             >RUN

    output             B A S I C
                        ! ! !
             >
```

1. HOW TO DISPLAY CHARACTERS BY USING POKE

In addition to using the CHR$ function in a PRINT statement, there is another way to display characters on the screen; you can use the POKE function to directly change the contents of memory. Characters to be displayed on the screen are stored in a block of memory specially allocated for that purpose. Every potential character location on the screen has a memory address in that block associated with it. If you place a character into a memory address associated with a screen location, that character will appear on the screen in that spot.

1.1 Video Screen Memory Addresses

The block of memory set aside to hold the characters being displayed on the screen begins with adress 6144 (decimal). The first screen location-- the upper left corner of the screen-- is associated, therefore, with memory address 6144. The second screen location-- the location just to the right of the first-- is associated with memory address 6145. And so on. Because the screen is 64 characters wide, the last character location on the first line (upper right corner) is associated with address 6207. The first location on the SECOND line of the screen is 6144 + 64, or 6208. The screen contains 1024 locations, so the last screen location (the lower right corner of the screen) is associated with memory address 7167. When you use POKE, make sure the memory address you give is equal to or greater than 6144 and les than or equal to 7167. Otherwise you will be putting your characters into another part of memory, not the part associated with the video screen. The results could be disastrous.

1.2 Using POKE

When you used the PRINT statement, you used the ASCII code exactly as it is given in the table below (e.g. PRINT CHR$(65)). When you use the POKE statement, however, you must add 128 to the ASCII code. For an explanation of why this is necessary, see Appendix F, The System 88 Graphics Characters, in the System 88 User's Manual.

Let's say that you want to display a capital A in the first screen location. The POKE function takes the form POKE address,expression. Type:

```
POKE 6144,65+128
```

You can also simply say:

```
POKE 6144,193.
```

You can also use POKE to display the System 88 graphics characters on the screen. For information on the graphics characters, see Appendix F, System 88 Graphics Characters, in the System 88 User's Manual.

2. CHART OF BASIC CHARACTERS

2.1 Control Characters

NUL -- 0	DC1 -- 17
SOH -- 1	DC2 -- 18
STX -- 2	DC3 -- 19
ETX -- 3	DC4 -- 20
EOT -- 4	NAK -- 21
ENQ -- 5	SYN -- 22
ACK -- 6	ETB -- 23
BEL -- 7	CAN -- 24
BS -- 8	EM -- 25
HT -- 9	SUB -- 26
LF -- 10	ESC -- 27
VT -- 11	FS -- 28
FF -- 12	GS -- 29
CR -- 13	RS -- 30
SO -- 14	US -- 31
SI -- 15	SP -- 32
DLF -- 16	DEL -- 127

2.2 Numbers and Letters of the Alphabet

0	--	48	V	--	86
1	--	49	W	--	87
2	--	50	X	--	88
3	--	51	Y	--	89
4	--	52	Z	--	90
5	--	53	a	--	97
6	--	54	b	--	98
7	--	55	c	--	99
8	--	56	d	--	100
9	--	57	e	--	101
A	--	65	f	--	102
B	--	66	g	--	103
C	--	67	h	--	104
D	--	68	i	--	105
E	--	69	j	--	106
F	--	70	k	--	107
G	--	71	l	--	108
H	--	72	m	--	109
I	--	73	n	--	110
J	--	74	o	--	111
K	--	75	p	--	112
L	--	76	q	--	113
M	--	77	r	--	114
N	--	78	s	--	115
O	--	79	t	--	116
P	--	80	u	--	117
Q	--	81	v	--	118
R	--	82	w	--	119
S	--	83	x	--	120
T	--	84	y	--	121
U	--	85	z	--	122

2.3 Special Symbols

!	--	33	?	--	63
"	--	34	@	--	64
#	--	35	[	--	91
$	--	36	\	--	92
%	--	37	]	--	93
&	--	38	^	--	94
'	--	39	_	--	95
(	--	40	`	--	96
)	--	41	{	--	123
*	--	42	|	--	124
+	--	43	}	--	125
,	--	44	~	--	126
-	--	45	√	--	153
.	--	46	→	--	154
/	--	47	←	--	155
:	--	58	↑	--	156
;	--	59	÷	--	157
<	--	60	Σ	--	158
=	--	61	≈	--	159
>	--	62			

2.4 Greek Letters

α	--	128	β	--	129	γ	--	130
δ	--	131	ε	--	132	ζ	--	133
η	--	134	θ	--	135	ι	--	136
κ	--	137	λ	--	138	μ	--	139
ν	--	140	ξ	--	141	ο	--	142
π	--	143	ρ	--	144	σ	--	145
τ	--	146	υ	--	147	φ	--	148
χ	--	149	ψ	--	150	ω	--	151
Ω	--	152						

Appendix E

INTERFACING WITH ASSEMBLY-LANGUAGE PROGRAMS AND MEMORY

This section is written for those who want to interface assembly language programs with PolyMorphic Systems BASIC. It will also be of help to those who want to change the defaults for certain features in Poly BASIC. This discussion assumes an understanding of the front panel mode of operation for examining and changing the contents of memory locations. For information about the front panel mode, see the User's Manual, Appendix E: The Monitor; Front Panel Display.

1.1 ASSEMBLY LANGUAGE INTERFACE: CALL

The CALL function is used to invoke assembly language routines. The format is either CALL (addr,val) or CALL (addr) where both addr and val are expressions that must evaluate to 0<=addr<= 65535. The expression shown as "addr" is the address of the subroutine to be called. If "val" is present, it is passed to the subroutine in register pair HL. When the subroutine exits by issuing a RET, or conditional return instruction, the value in register pair HL will be converted to an integer and passed to the BASIC program as the value of the call.

The CALL function may also be used to invoke an assembly language overlay. (See the System 88 System Programmer's Guide for a discussion of overlays.) The correct syntax is:

```
CALL("abcd",A,B,D,H)
```

where abcd is the name of the overlay. A, B, D, and H are expressions that BASIC will evaluate and pass to the overlay in registers A, BC, DE, and HL respectively. ALL parameters must be given to the CALL function when invoking an overlay.

As with the normal CALL, the value in HL is returned to BASIC as the value of the CALL.

1.2 MEMORY EXAMINATION AND MODIFICATION: PEEK AND POKE

Note: modification by use of the POKE statement of areas of memory containing BASIC, BASIC programs or data, or the system core may result in anomalous program behavior, possibly resulting in the loss of the program and/or its data.

1.2.1 PEEK

The PEEK function takes the form PEEK addr, val where addr is an expression evaluating to the range 0<=addr<=65535 as a memory address, and returns the integer contents of that memory location.

Using PEEK on areas of the address space not populated with memory may give anomalous, possibly non-repeatable results.

1.2.2 POKE

The POKE statement takes the form POKE addr, val where addr is an expression evaluating to the range Ø<=addr<=65536 for the memory address to modify, and Ø<=val<=255 for the 8 bit quantity to store at that address. As noted above, exercise caution when using the POKE statement.

1.3 ACCESSING THE I/O PORTS: INP AND OUT

The 8Ø8Ø processor IN and OUT functions can be performed through BASIC using the INP function and the OUT statement respectively. The format of the INP function is INP (port), where port is the port address with a value Ø<=port<=255. INP (port) returns as an integer the eight-bit status resulting from an IN instruction to the desired port. Note that INP(Ø) through INP(31) are reserved for system use, and that INP of an undefined port may give anomalous results. The format of the OUT statement is OUT port, val where port is the 8Ø8Ø port address with a value Ø<=port<=255 as in INP above, and val is the eight-bit value Ø<=val <=225 that is sent to the specified port. Note that ports Ø-31 (decimal) are reserved for system use, and that issuing an OUT to a system-controlled device or port may result in anomalous behavior, possibly resulting in the loss of the program and/or its data.

1.4 ACCESSING THE TYPE-AHEAD BUFFER: INP(Ø), INP(1), INP(2), and OUT Ø

Calls to INP with port addresses Ø-2 return data regarding type-ahead. INP(Ø) returns the status of the type-ahead buffer; Ø if the buffer is empty, and not Ø if there is at least one character in the input buffer. INP(1) returns the next character as an integer (ASCII) value, without echoing it to the screen, and INP(2) returns the next character as an integer and echoes the character to the screen. The statement OUT Ø, val places the ASCII character with integer value val into the input buffer. It should be noted that an attempt to put characters into the input buffer when it is full will be ignored. Printing a Control-X character will flush the input type-ahead buffer.

1.5 RE-ENTERING BASIC FROM FRONT PANEL DISPLAY

To re-enter BASIC from the front panel display, type SPJ32ØØ for "cold start" (BASIC assumes there is no program in effect); type SPJ32Ø3 for "warm start" (BASIC assumes there is a program in the machine). Then type carriage return and G. The above operations set the program counter to the specified address.

Example:

```
  enter:100 REM This program uses OUT 0 to list and scratch
        110 REM itself....
        120 REM also demonstrates use of multiline functions
        130 REM and dummy arguments.
        140 Z=FNI("LIST")+FNI("SCR")
        150 STOP
        160 REM Function to stuff string into input buffer
        170 REM followed by a carriage return.
        180 DEF FNI(S$)
        190 FOR I=1 TO LEN(S$)\S1$=MID$(S$,I,I)C=ASC(S1$)\OUT 0,C\NEXT
        200 OUT O,13\RETURN 0
        210 FNEND
        >RUN

        Stop in line 150
        >>LIST
        100 REM This program uses OUT 0 to list and scratch
        110 REM itself....
        120 REM also demonstrates use of multiline functions
        130 REM and dummy arguments.
        140 Z=FNI("LIST")+FNI("SCR")
        150 STOP
        160 REM Function to stuff string into input buffer
        170 REM followed by a carriage return.
        180 DEF FNI(S$)
        190 FOR I=1 TO LEN(S$)\S1$=MID$(S$,I,I)C=ASC(S1$)\OUT 0,C\NEXT
        200 OUT O,13\RETURN 0
        210 FNEND
        >>SCR
        >LIST
        >
        >
```

Appendix F

COMMANDS, STATEMENTS, FUNCTIONS, AND KEYWORDS IN DISK-BASIC

Next to each entry are the page numbers where you can turn for information about that topic.

INDEX

```
5 S=0¢REM SHAPE
7 PRINT CHR$(12)¢Z=RND(TIME(1)/2 16)
10 X=RND(1024)+6143
20 Y=RND(8)
30 ON Y GOSUB 110,120,130,140,150,160,170,180
40 IF X<6144 THEN X=6144 ELSE 50
44 IF X>7167 THEN X=7167 ELSE 50
50 S=0¢P=PEEK(X)¢IF P=0 THEN S=63
52 POKE X,S¢GOTO 20
110 X=X-64¢RETURN¢REM UP
120 X=X+64¢RETURN¢REM DOWN
130 X=X+1 ¢RETURN¢REM RIGHT
140 X=X-1 ¢RETURN¢REM LEFT
150 X=X-63¢RETURN¢REM UP AND RIGHT
160 X=X-65¢RETURN¢REM UP AND LEFT
170 X=X+65¢RETURN¢REM DOWN AND RIGHT
180 X=X+63¢RETURN¢REM DOWN AND LEFT
RUN
```