BASIC: A MANUAL

Written by

Robin C. Soto, R. T. Martin
and Robert Scott Keeney

PolyMorphic
Systems

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

TABLE OF CONTENTS

Section 1

INTRODUCTION

You are about to learn a very simple language. You will never speak a word of BASIC to any human being. But the things you can do with this language make it possible for you, with the help of your computer's "brain", to develop programmed information with a high degree of speed and reliability.

BASIC was originally developed in 1963 at Dartmouth College by Professors Kemeny and Kurtz, who conceived of BASIC as a computer language simple enough to be used by beginners, yet powerful enough to carry out sophisticated computation.

BASIC is a machine language "interpreter" which the user may develop BASIC programs. BASIC machine language is "loaded" in the computer. The computer then "understands" programs written in BASIC.

The user takes a problem and a definition of the problem to the computer and develops a BASIC program. With a BASIC program, the user defines the problem and the methods for its solution once only, without having to repeat the process during subsequent computations. The computer, using the program, accumulates, stores and organizes the needed information, keeping in mind the ways to solve the problem and the problem's definition.

A BASIC program is not a static accumulation of words and symbols (even though a program does accumulate information). A program is a dynamic process, somewhat like the continually moving parts of mobiles. A program is built out of parts which go together to form an interpenetrating construction. Your BASIC manual is designed with that principle in mind, by providing the user with a careful development of all the BASIC parts required to begin knowledgeable construction of a BASIC program.

1.1 MANUAL CONTENT

BASIC: A Manual has been written to provide BASIC users at every stage of programming proficiency with a sufficient and plainly set-forth body of information. Basic information has been grouped into sections, each section building upon information provided in previous sections, so that the novice user may develop, section by section, a coherent sense of BASIC and its potential. If you do not understand some aspect in an early section of this manual, it will be clarified by the information contained in a later section. This manual has also been designed to permit quick, complete referencing by the most advanced user. The manual is arranged in 10 sections with several appendices containing supplementary material. The next sections are:

Section 2 -- Getting Into BASIC: This section deals with the primary elements of a BASIC program, such as deletion and correction techniques and carriage return, and discusses direct statements.

Section 3 -- Inputting Your Program: Section 3 deals with the actual typing of 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 may use when you run your BASIC program.

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

Section 6 -- Functions and Subroutines: This section discusses functions intrinsic to Poly 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 then in BASIC.

Section 8 -- The PLOT Feature: The Poly 88 BASIC PLOT feature is described and demonstrated in this section.

Section 9 -- Error Messages Generated by BASIC: A list of error messages generated by BASIC, along with possible causes for those messages.

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

Appendix A -- Loading BASIC and Saving and Loading a BASIC Program: The proper methods for saving and loading BASIC programs and for loading BASIC itself.

Appendix B -- Sample Programs: This appendix contains sample programs which demonstrate various aspects of computer programming pertinent to your particular Poly 88 system.

Appendix C -- The BASIC Character Set: The character set for your Poly 88 BASIC is given in this appendix.

Appendix D -- Interfacing with the Assembler and Memory: This appendix discusses methods for interfacing BASIC and assembly programs. It also discusses procedures whereby the user may directly access memory.

Appendix E -- Commands, Functions and Keywords Recognized by BASIC: A list of all commands, statements, functions and keywords to be found in BASIC is given.

1.2 THE EXAMPLES IN THIS MANUAL

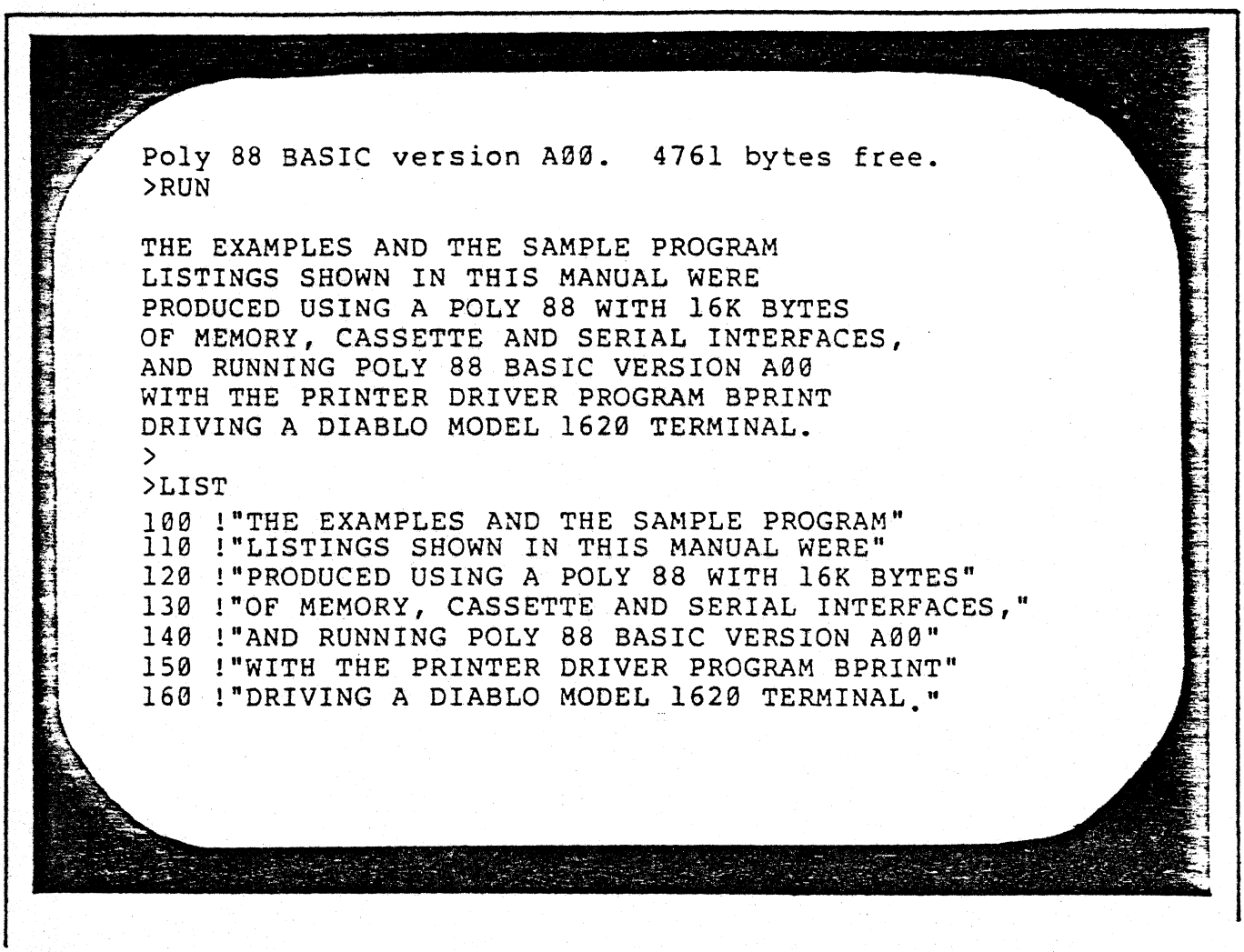
The examples in this manual were typed on a Diablo Hy-Type 1620 Terminal linked to a Poly 88 computer. Hence, the examples represent actual computer printouts and will resemble the characters put out on the video screen. Try the examples given with each section and many aspects of BASIC which are not clarified at once in the text may become clear to you through the actual process of entering-in the examples on the keyboard.

In most of the examples, "enter" is used across from the first line of the example. The information located on the line across from "enter" should be typed in by the user as it appears in the example.

That section of the example marked "output" indicates the computer's response to the "enter" section. When the "enter" section of the example has been typed in correctly by the user on the computer's keyboard, type a "carriage return" 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, page 8.

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. Everything on a program line after the word REM will be ignored by BASIC, except to be reproduced when the program is displayed. The comments after the REM's appearing in the examples are designed to help clarify the examples for you.


Poly 88 BASIC version A00. 4761 bytes free.
>RUN
THE EXAMPLES AND THE SAMPLE PROGRAM
LISTINGS SHOWN IN THIS MANUAL WERE
PRODUCED USING A POLY 88 WITH 16K BYTES
OF MEMORY, CASSETTE AND SERIAL INTERFACES,
AND RUNNING POLY 88 BASIC VERSION A00
WITH THE PRINTER DRIVER PROGRAM BPRINT
DRIVING A DIABLO MODEL 1620 TERMINAL.
>
>LIST
100 !"THE EXAMPLES AND THE SAMPLE PROGRAM"
110 !"LISTINGS SHOWN IN THIS MANUAL WERE"
120 !"PRODUCED USING A POLY 88 WITH 16K BYTES"
130 !"OF MEMORY, CASSETTE AND SERIAL INTERFACES,"
140 !"AND RUNNING POLY 88 BASIC VERSION A00"
150 !"WITH THE PRINTER DRIVER PROGRAM BPRINT"
160 !"DRIVING A DIABLO MODEL 1620 TERMINAL."

Section 2

GETTING INTO BASIC

2.0 SOME BASIC FUNDAMENTALS

Have you loaded BASIC? Appendix A will show you the right way to load BASIC into your machine, so that the machine will be able to "talk" with you in BASIC. In this process you will make arrangements with the computer and give it BASIC to store in its "brain".

After BASIC is properly loaded into your machine, BASIC will display a message telling you which version of BASIC has been loaded, and will tell you that it is ready to listen to you by displaying a prompt symbol (>) at the left hand side of your monitor screen.

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

2.1 THE KEYBOARD AND DISPLAY

The computer keyboard works much like a standard typewriter. There is a shift key on the keyboard which functions like a typewriter shift key. However, most keyboards have only upper-case letters and the shift key is used for the symbols on the upper case above the numbers and for some special symbols.

The character for the keys you depress will appear on the video display. The space bar functions exactly like a typewriter space bar, save that it makes one blank space on the screen.

2.1 A. Giving Instructions to BASIC

There are two major ways in which you may give BASIC some simple instructions. The first of these two methods is called a Direct Statement. 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 B. 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 C. Interrupting BASIC

To interrupt any process in BASIC, use the Control-Y command. To make a 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 stop execution and return with a prompt.

2.1 D. What To Do If You Make A Mistake

BASIC has several methods of dealing with mistakes made while inputting an instruction. The table below summarizes the deletion commands available in BASIC:

To delete:

Individual characters:	Use the DEL or RUBOUT key to backspace the number of spaces 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 line:	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. Other elements of BASIC instructions and program lines are discussed in following sections of this manual.

2.2 A. 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 B. Arithmetic Operators

BASIC executes arithmetic operations in response to the following symbols, and, if several are used in the same expression, in the order listed:

Example	Symbol	Operation
>PRINT 9^3 729 >	↑	Exponentiation. On keyboards without this symbol a Shift-N is used.
>PRINT 7*9 63 >	*	Multiplication
>PRINT 234.56/.8904 263.43217 >	/	Division
>PRINT 23.89 + 67.08 90.97 >	+	Addition
>PRINT 567.9-56.12 511.78 >	-	Subtraction

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 the 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 C. 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

Symbol	Operation
≧ =>	is greater than or equal to
≦ =<	is less than or equal to

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

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

                    >
           enter    >PRINT 7>7
           output    Ø
                    >

                    >
           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>=Ø THEN PRINT X ELSE PRINT "INPUT POSITIVE NUMBER"
         INPUT POSITIVE NUMBER
```

2.2 D. Logical Operators

BASIC can solve problems in Boolean logic using the following three operators: AND, OR, and NOT. The result of a Boolean operation is always a 1 (if true) or a Ø (if false).

```
                    >
      Example:      >PRINT 1 AND 1
                     1
                    >
                    >
                    >PRINT 1 AND Ø
                     Ø
                    >
                    >
```

BASIC will also check the validity of a Boolean statement, returning a 1 (is true) or a Ø (if false).

```
Example:          >
          enter   >PRINT (1 AND 1)=(1 AND NOT1)
          output   Ø
                  >
                  >
```

2.2 E. 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, or (4) expressions..

2.2 F. Constants

A constant is a number representing an unvarying quantity. When BASIC stores a number in memory, it represents it with a maximum of eight digits plus an exponent. Therefore all numbers larger than eight digits are rounded off by BASIC. This means that when BASIC adds the two numbers 5ØØØØØØ + .009, it will return with the incorrect answer of 5ØØØØØØ. In order to represent numbers larger than 99,999,999 BASIC uses the exponential notation (or scientific notation) form (number X $1Ø^{63}$).

Examples:

3.76E+Ø2	means	+3.76 X $1Ø^{Ø2}$	(+3.76 X 1ØØ), or +376
-3.76E+Ø2	means	-3.76 X $1Ø^{Ø2}$	(-3.76 X 1ØØ), or +376
3.76E-Ø2	means	+3.65 X $1Ø^{-Ø2}$	(+3.76 X .Ø1), or +.Ø376
-3.76E-Ø2	means	-3.76 X $1Ø^{-Ø2}$	(-3.76 X .Ø1), or -.Ø376

2.2 G. 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 IS A STRING (1+1*(3+SQRT(16)))TOO"

2.2 H. 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 I. 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
```

2.3 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). Direct statements are usually 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 3*(50/25)
          output  6
                >
```

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

it may take this 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 64 characters. Each program line is given 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. At that time BASIC will execute the program lines in numerical order. This section deals with the actual typing in of your BASIC program. It contains information about line numbers, and program lines.

3.1 PROGRAM LINE NUMBERS

Every program line begins with a line number which must be an integer ranging from Ø to 65535, inclusive. Any line of text typed to BASIC which begins with a number is processed by the editor as a program line. Blanks or tabs before the line number are ignored by BASIC, and the first blank or nondigit that follows a line number terminates that line 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 Ø 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 no other action is taken by BASIC on that line.

The techniques for adding, replacing and deleting 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 carriage return. If a new program line contains only a program line number, BASIC will delete any pre-existing program line beginning with that same program line number.

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    >RUN

output     6
          >
          >
```

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 transform program execution to another program line. Branches can be conditional, dependent upon a test condition

```
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.

or unconditional. Go to section 5, for examples of branching statements.

Section 4

RUNNING YOUR PROGRAM

4.0 CONTROL COMMANDS

Now that you have learned how to set up a program, you want to know how to run it, too. 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 A--Loading BASIC, and Saving and Loading a BASIC program.

4.1 LIST

The list command is used when the programmer wishes 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 the line numbers are not supplied, the entire program is displayed. If the first line number is provided, 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. 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                        enter  >LIST 15,25
output  10 REM SAMPLE PROGRAM       output  15 X=1
        15 X=1                              20 Y=2
        20 Y=2                              25 PRINT X+Y
        25 PRINT X+Y                        >
        >
```

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

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
                >
                >
```

4.2 REN (Renumber)

After you have made many insertions in a program, the line numbers often 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, and 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 a line number of 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:

```
>                                >
>                                >REN
>10 REM SAMPLE PROGRAM           >LIST
>12 INPUT X                      10 REM SAMPLE PROGRAM
>70 PRINT X+1                    20 INPUT X
>                                30 PRINT X+1
                                 >
>                                >
>REN 50                          >REN 100,100
>LIST                            >LIST
50 REM SAMPLE PROGRAM            100 REM SAMPLE PROGRAM
60 INPUT X                       200 INPUT X
70 PRINT X+1                     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 t 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 the following paragraph 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. The first thing that is done is to clear the variable and string areas. This means;

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

2) that all strings are set to null (length of zero), and
3) unless initialized by a DIM statement, both strings and vectors (arrays) will take on the default size of 1Ø elements.

Next, 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.

The pointer used to access DATA statements for READ (see section 5.2 b 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 if 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. Then BASIC begins executing the program at either the line number specified with the RUN command, or at the first line of the program.

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, the continue command, CON, must be used.

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 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, the program execution begins at the statement after the STOP statement. When the CON command is used to continue after an interruption caused by 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

After program execution it is often necessary to "clear" all variables and strings: that is to reset them to their original initialization within the program. This avoids any possible cumulative effects of executing a program more than once. Use of the CLEAR command sets all input variables to Ø, and all input strings to a null value.

4.7 SCR (Scratch)

The command SCR, typed after a prompt, erases all information in working memory; your program and its data.

4.8 Summary of Control Commands

CLEAR	Resets all input variable values to ϕ 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. Returns a prompt to the user.
LIST	Lists program. Takes the form, LIST <optional line number>, <optional line number>.
REN	Renumbers program lines. Takes the form REN optional beginning value>, <optional increment value>.
RUN	Begins execution of a program either at the beginning of the program or at the optionally supplied line number. It takes the form, RUN <optional line number>.
SCR	Erases the program, and anything else typed from the terminal.

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 B--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 Statements, LET.

5.1 A. 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 everything to the right of it, including the letters "REM" are ignored by BASIC. BASIC will, however, print the REM statement when the program is listed. The REM statement, unless it is the first statement on the program line, must be preceded by a back-slash (\).

5.1 B. STOP

The STOP statement is inserted in a program whenever a permanent or recoverable halt is desirable. To continue execution from a STOP, use the continue command, CON described in section 4.5.

5.1 C. Assignment Statement (LET)

An assignment statement is used to set a variable to a given value or expression. The usual form is <variable>= <constant, variable or expression>, for example: A=19. Using this example, the variable "A" is set equal to 19. The expression on the right can be more 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$+B$
          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 takes up unnecessary space on a line. The following examples are identical in meaning.

Example:

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

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 A. 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 of the program.

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 no question mark is printed by BASIC. If more than one variable is asked for in one input statement, they must be separated by commas.

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 B. 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 C. 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>. Without the optional line number, the READ statements would be directed to begin reading data from the first DATA statement in the program. With the line number included, the READ statements would be 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
          >
```

Example (continued):

```
                    >
          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 D. Single Character Input Functions INP (Ø), INP (1), INP (2)

The functions INP (Ø), INP (1), and INP (2) allow the user to test for characters in the input buffer, and input single characters from the keyboard. The function INP (Ø) returns Ø 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; INP (2) returns the integer value of the next character from the keyboard buffer and echoes it to the screen (See appendix C for decimal values for the character set).

Example: enter

```
100 REM DEMONSTRATE INP(Ø) TESTING FOR INPUT
110 PRINT "YOU HAVE 10 SECONDS TO TYPE COW"
120 PRINT "?",
130 Z=TIME(Ø) \ REM RESET CLOCK
140 IF INP(Ø)>Ø 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.
>
```

(Note: characters are stored inside the computer as numbers. See Appendix C, the BASIC Character Set.)

Example:

```
100 REM USE INP(2) TO FIND DECIMAL VALUES OF CHARACTERS
110 PRINT "TYPE A CHARACTER, AND I'LL TELL YOU ITS VALUE"
120 PRINT "?",
130 A=INP(2)\PRINT " IS A DECIMAL",A
140 GOTO 110
>RUN

TYPE A CHARACTER, AND I'LL TELL YOU ITS VALUE
?A IS A DECIMAL 65
TYPE A CHARACTER, AND I'LL TELL YOU ITS VALUE
?H IS A DECIMAL 72
TYPE A CHARACTER, AND I'LL TELL YOU ITS VALUE
?7 IS A DECIMAL 55
TYPE A CHARACTER, AND I'LL TELL YOU ITS VALUE
? IS A DECIMAL 7
TYPE A CHARACTER, AND I'LL TELL YOU ITS VALUE
? (Control-Y command used here)

Interrupted in line 130
>>
```

5.3 OUTPUTTING DATA

There are several ways of changing 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 A. 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 --see next page--
```

Example (continued):

```
output  RUBBER CHICKEN 10 4
        SECOND LINE
        >



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

output  RUBBER CHICKEN 10 4
        SECOND LINE
        >
```

In order to save space on the program line, the word PRINT may be abbreviated to an exclamation mark symbol (!), as in the above example. 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 of 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 B. Formatting the PRINT Statement

If no formatting is specified 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 Section 2.2 F, Constants).

1. Numbers less than or equal to eight digits in length and in non-exponential form will be printed as given.

```
Example:
                >
     enter      >PRINT 12.34567
    output       12.34567
                >
```

2. Numbers greater than eight digits in length 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 less than or equal to eight digits in length 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 greater than eight digits in length are rounded off and printed in non-expo-

nential 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 that distance across the screen before printing the elements in the print list. The TAB value must be less than 256 and positive.

Example:

```
          >
 enter    >10 PRINT TAB(15),"UNIT ONE",TAB(25),"UNIT TWO",
          >20 PRINT TAB(35),"UNIT THREE"
          >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 manner in which numerical data may be outputted by a print statement. A format string may appear anywhere in a PRINT statement after the PRINT command, and must begin with a per cent symbol (%). An empty format string will allow 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, 34544567.
```

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.

Example:

```
enter   10 !\!"IN NEW DEFAULT FORMAT--"
        20 PRINT %$C#,9999
        30 FOR I=2000 TO 5000 STEP 1000
        40 PRINT TAB(30),I,
        50 NEXT
        60 !\!"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, <n>F<m>.

```
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, <n>I.

```
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 %1ØE3,3798.678892
output   3.799E+Ø3
        >
```

Note: The number 3.799E+Ø3 represents 3.799 X $1Ø^3$.
(For a further discussion of scientific notation, or exponential form, see Section 2.2 F, Constants).

Example:

```
        >
 enter  >PRINT 3.799E+Ø3
output   3799
        >
```

In order to avoid format specification errors, it is important to remember to reserve enough space in the print field by use of a large enough n so that the number given to the format specification may be printed. For example, 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,+,Ø,2)= 11 spaces); otherwise an error message is generated.

Example:

```
        >
 enter  >PRINT %1ØE5,234.56
output  Format error
 enter  >PRINT %11E5,234.56
output  2.34560E+Ø2
        >
```

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 perhaps, for a certain set of arguments.

Let's say that we wanted to print the integer from 1 to 1Ø inclusive, and their squares. We could write a BASIC program that would execute this process, and would look 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 11Ø. We then see if J is greater than 1Ø. The first time through, J has the value of 1, so we continue execution with line 13Ø, 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 12Ø. We "loop" through lines 12Ø, 13Ø, 14Ø and 15Ø until J is incremented by line 14Ø to the value 11. Then, when we perform the IF statement on line 12Ø, J is greater then 1Ø, so we go to line 16Ø 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 1Ø 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 11Ø.

3) A terminating condition; in the example, the loop will terminate, or stop, when J is greater than 1Ø, as detected by the IF statement in line 12Ø.

4) An increment (or decrement) to apple to the loop variable: In the example on line 14Ø, 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 1Ø.

5) A set of statements that are executed repeatedly, also called the loop body. In the example, the loop body consists of the single PRINT statement on line 13Ø.

6) An indicator marking the end of the loop. In the example, the GOTO 12Ø statement on line 15Ø denotes the end of the loop. When the variable J exceeds the terminating condition, 1Ø, as specified by the IF test on line 12Ø, program execution will resume past the end of the loop, at line 16Ø.

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:

```
>
>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
```

Let's 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 110, the characters "TO 10." This gives the terminating value to test the loop variable (J in this case) as 10, just as it did in the IF statement on line 120 of the other program. The terminating value, in this case the number 10, 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 140, 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 us 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 integer, or positive. For example, the statement

```
1ØØ FOR W=-1 TO -2Ø 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 indefinetely, 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 A. Nesting of FOR-NEXT Loops

Often we would 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 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 DONT 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     >110 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,
          >
```

Examples:

```
        >
 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 other program lines. 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 A. GOTO

The GOTO statement allows the programmer to 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 with a Control-Y command.

5.5 B. 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, the first line number listed after the GOTO will be

jumped to by BASIC. 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 !"X EQUALS ONE"
      >40 GOTO 80
      >50 !"X EQUALS TWO"
      >60 GOTO 80
      >70 !"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 equal 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 !"YOU'RE CLOSE"\GOTO 60
      40 !"YOU'RE WARMER"\GOTO 60
      50 !"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 C. IF-THEN

The IF-THEN statement is used to set up a test condition which must be met before further instructions 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, string to 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 !"HERE ARE INSTRUCTIONS..."\GOTO 60
      >50 !"O.K. CATCH YOU LATER"
      >60 REM END OF PROGRAM
      >RUN
```

output -- see next page

```
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
enter   >10 INPUT "GIVE ME A NUMBER--",X
        >20 IF X=1 THEN !"RIGHT ANSWER"\!"GO ON!"\GOTO 200
        >30 !"X NOT EQUAL TO ONE"
        >200 !"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 D. ELSE

An IF-THEN statment 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
        >
```

Example:

```
>
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 !"GO!" ELSE !"WRONG"
       >RUN

       GO!
       >
       >
```

5.5 E. 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 SUMMARY OF PROGRAM STATEMENTS

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.

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 next PRINT or INPUT statement appears on the same line as original input.

LET — Optional assignment statement. LET<variable>=<variable, expression, or string>.

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>.

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 B) or a TAB (Section 5.3 B) 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, and preceded by a back-slash unless it is the first statement on the line. REM<comments>.

RESTORE — Used to change the order that 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.

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 more efficiently structuring your program. These are: 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 A. Regular Intrinsic Functions

SQRT(expression) . Returns the positive square root of a positive expression. An expression less than Ø 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.

COS(expression) Returns the cosine of the expression in radians.

SIN(expression)	Returns the sine 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 Ø 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 Ø but less than 1. If the seed value is Ø 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(Ø) 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 .164Ø625.

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.

To produce random numbers greater than number A and less than number B, the following expression should be used: (RND(Ø)*(B-A))+A

The RND function is often used in combination with the INT function to produce random integers. The statement INT(RND(Ø)*6)+1 simulates the roll of one die, giving numbers between 1 and 6 inclusive.

Example:

```
      >
enter >LIST
     ·IØØ REM SIMULATION OF THROWING ONE DIE
      11Ø Z=RND(TIME(1)/65536)\ REM RANDOMIZE
      12Ø FOR I=1 TO 1Ø
      13Ø D=INT(RND(Ø)*6)+1\ REM DIE VALUE SUCH THAT Ø<D<7
      14Ø PRINT "YOUR THROW IS",D
      15Ø NEXT
      >RUN

      YOUR THROW IS 2
      YOUR THROW IS 5
      YOUR THROW IS 6
      YOUR THROW IS 1
      YOUR THROW IS 5
      YOUR THROW IS 5
      YOUR THROW IS 4
      YOUR THROW IS 4
      YOUR THROW IS 2
      YOUR THROW IS 3
      >
```

TIME (expression) The TIME function returns as its value the 16 bits of the POLY 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 B. Sample Programs, for examples.

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

6.1 B. Intrinsic Functions Directly Accessing Memory and the 8080 System

(See Appendix D, 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. Ports 0 through 31 (decimal) are reserved for the system. The statement !INP (80) tells

you what value is in the 8Øth port of the Poly 88.

FREE(Ø) — !FREE(Ø) prints the number of bytes available in memory.

OUT 8Ø8Ø port, expression — This instruction allows the programmer to perform an 8Ø8Ø OUT instruction to a specified port. For example, OUT 4Ø,3 performs an OUT 4Ø instruction with 3 in the 8Ø8Ø accumulator. Ports Ø though 31 (decimal) are reserved for the system.

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

PEEK(memory byte) — This function allows the programmer to examine the value being held in the specified memory byte location. For example, !PEEK(3ØØØ) will tell you what value is in memory byte 3ØØØ.

6.1 C. 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(A
      >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
          >
```

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
          >
```

6.2 USER-DEFINED FUNCTIONS

BASIC allows programmers to define their own 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. Therefor, when the function is called upon during program execution, the arguments of the function call are substituted in for the dummy arguments 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 !"USE CONTROL-Y TO EXIT"
20 DEF FNS1(A,B)=A+B
30 INPUT1 "GIVE 2 NUMBERS--",X,Y
40 !" THEIR SUM IS: ",FNS1(X,Y)
50 !" 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)
Interrupted 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 fron 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 !"  SQUARE ROOT OF YOUR"
        >210 !"NUMBER IS: ",SQRT(X)
        >220 RETURN
        >RUN

output  GIVE POSITIVE #: 356  SQUARE ROOT OF YOUR
        NUMBER IS:  18.867963
        Stop in line 40
        >>
```

Care should be taken that program execution not be allowed to "fall into" the subroutine. For example, in the above program, if the STOP statement at line 40 is removed, the subroutine is executed 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 !"  SQUARE ROOT OF YOUR"
        210 !"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 have been incorporated into one section in this manual because, in many ways, a string can be treated in the same manner as an array. Both strings and arrays consist of a series of elements, which may be indexed by the use of subscripts.

7.1 ARRAYS

An array is a list of numerical 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 Ø (written X(Ø)). Note that subscripts denoting a position in an array begin with Ø. The second item would be referenced by the subscript 1 (X(1)), and so on. The subscripts may, in turn be represented by a variable (X(I)).

Example:

```
>
>LIST
enter  10 REM PRINT OUT ARRAY IN REVERSE ORDER
       20 X(Ø)=1Ø\X(1)=2Ø\X(2)=3Ø\X(3)=4Ø\X(4)=5Ø
       30 FOR I=4 TO Ø STEP -1
       40 PRINT X(I)
       50 NEXT
       >RUN

output  5Ø
        4Ø
        3Ø
        2Ø
        1Ø
       >
```

If an array is not assigned a certain length within the program, it is assumed that it consists of one dimension, and not more than 1Ø elements. To reserve more space than this in memory, the dimension statement is used. This takes the form, DIM<variable array name>(number of items). For example, DIM X(5ØØ). 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 =	Ø	1	2	3
I =	Ø	1Ø	11	12	13
	1	14	15	16	17
	2	18	19	2Ø	21
	3	22	23	24	25

The position X(3,2) contains the number 24. A sample program to print this array would be:

Example:

```
          >
enter     >10 DIM X(3,3)
          >20 FOR I=0 TO 3\FOR J=0 TO 3
          >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.

7.2 STRINGS

A string is a list of characters (such a list may also contain 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 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 ($) symbol; such as "A1$."

Strings may be dimensioned to a particular length by use of the DIM statement. Unlike arrays, strings may consist of only one dimension. If no length is assigned to the string, room is reserved for only 1Ø characters (including blanks). Any string consisting of more than 1Ø characters is truncated to 1Ø characters unless a DIM statement is used. The amount of space reserved by a DIM statement is limited only by available memory space.

The dimension statement for a string takes the form, DIM <string variable> (number of charcters). For example, DIM A$(3Ø), reserves space for 3Ø characters on the string A$. A string may be dimensioned only once within a program.

Referencing a string element by use of subscripts differs somewhat from the method used on arrays. When referencing string elements, subscripts begin at 1: i.e., the first character of string S$ is S$(1,1).

Example: Given string S$:

S$(J) refers to the substring beginning at character position J through to the end of the string.

S$(J,K) refers to the substring beginning at character position J through character position K.

S$(J,J) refers to character at position J.

It is possible to concatenate substrings and strings using the additional symbol, +. If the conbined strings or substrings are larger than allowed by the program DIM statements, they will be truncated to fit.

Examples:

```
          >
 enter    >1Ø REM STRING INDEXING
          >2Ø DIM T$(12)
          >3Ø T$="TACKY-"
          >4Ø !T$(3)\!T$(2,4)\!T$(3,3)
          >5Ø T$=T$+T$\!T$
          >RUN

 output   --see next page--
```

Example (continued):

```
output   CKY-
         ACK
         C
         TACKY-TACKY-
         >
```

Strings, substrings, 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 a INPUT statement, the input must not be surrounded by quotation marks. When strings are found in DATA statements, they must be surrounded by quotation marks.

Section 8
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, and is 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, the following form must be used, 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 demonstration of the PLOT feature, see Appendix B -- Sample Programs.

Section 9
ERROR MESSAGES GENERATED BY BASIC

If you make an error using direct statements, BASIC will respond with a simple error message. If an error is encountered during execution of the program statements, BASIC will reprint the program line in which the error occurred and point to the approximate point in the line containing the error. An error message will also be printed.

```
Example:    enter    >
                     >Y=3*(SQRT(16)+YCLEPT)
            output   Syntax error
                     >
            enter    >10 Y=3*(SQRT(16)+YCLEPT)
                     >RUN

            output   10 Y=3*(SQRT(16)+YCLEPT)
                                      ↑
                     Syntax error
                     >
```

The error messages that you might receive are listed below along with their possible causes.

9.1 ERROR MESSAGES

Arg mismatch error

Number of arguments in user-defined function definition was 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
                     >
```

Bad argument error

May occur if a parameter given to the PLOT function is out of bounds (for example, if X>127 or Y>47).

Can't continue

BASIC has been asked to continue execution of a program but cannot do so, either because no program exists, or because the end of the program has already been reached. BASIC also will not continue execution if a change is made in the program after an interruption, or if a CLEAR command has been used. After an interruption, BASIC indicates that it can continue with a double prompt (>>). If it cannot continue, BASIC returns after an interruption with a single prompt (>).

Checksum error

A checksum error is the result of a tape loading problem. When loading BASIC, a question mark may indicate a checksum error. When loading a BASIC program, a checksum error will be indicated by a checksum error message. A checksum error indicates either an incorrectly loaded program or tape damage of some kind.

Complexity error

An expression is too complex for BASIC to evaluate.

Control stack error

An internal stack has overflowed, possibly through using too many functions which call upon themselves.

Dimension error

Incorrect dimensioning. For example, redimensioning an array or string within a program, or using a variable as an argument in a DIM statement (i.e., DIM X (A)).

Division by zero error

An attempt was made to divide a variable or expression by Ø.

Double def error

An attempt was made to define a user-defined function twice within on program.

Format error

Several causes, all having to do with incorrect outputting of data. For instance, a format error may occur if an attempt is made to print out a number in the F-format in a field of greater than 25 spaces. Usual cause -- incorrect format string.

FOR-NEXT error

Happens if improper nesting of FOR-NEXT loops occurs. Other possible causes include incorrect loop index, NEXT variable, STEP value, loop index initial or terminal value, or mismatched FOR and NEXT variables.

Function def error

Attempt was made to use an undefined function.

Illegal direct error

Attempt was made to use a statement not acceptable as a direct statement. For example: (See section 2.2 -- Direct Statements)

```
          >
enter     >GOTO 1ØØ
output    Illegal direct error
          >
```

Input error--retype

An attempt was made to input a string where a number was asked for, or vice versa.

Length error

The last line entered exceeded 64 characters.

Line number error

An attempt was made to reference a non-existent program line.

Memory full error

No more memory space is available. May occur when infinite loop allowed to run uninterrupted. For example:

```
          >
enter     >10 GOSUB 10
          >RUN

output    10 GOSUB 10
                   ↑
          Memory full error
          >
```

Missing NEXT error

There are not enough NEXT statements in the program to match the FOR statements.

Out of bounds error

Possible causes include a program line number greater than acceptable (>65536), or an attempt to dimension an array larger than memory will hold (DIM X(5ØØØØ)).

Overflow error

An attempt was made to evaluate an expression too large for BASIC to represent. For example:

```
          >
enter     >PRINT 3*10^64
output    Overflow error
          >
```

READ error

Not enough data in DATA statement, or data was not in proper form (constants or variables, depending upon type of variable in READ statement).

RETURN without GOSUB error

A RETURN statement was found without an accompanying GOSUB statement in the program.

Subscript error

An attempt was made to use a nonexistent subscript, or a subscript larger than allowed by DIM statement. For example:

enter
```
>10 DIM X(5)\!X(20)
>RUN
```

output
```
10 DIM X(5)\!X(20)
                ↑
Subscript error
>
```

Syntax error

There are many, many possible causes for syntax error. In general, a syntax error is a typing error (i.e., PRIMPT X). Incorrect form of program statements is also a cause (i.e., IF X=0 GOTO 200 (no THEN)).

Type error

An attempt was made to use a string function on a numerical variable or vice versa. For example, PRINT SQRT (A$), attempts to use a numerical function on a string variable.

Verify error

This error may occur when verifying a BASIC tape. The error message indicates that the tape is invalid: the program in memory has been changed, the tape has been incorrectly saved, or the tape has been damaged.

Section 10

OPTIMIZING YOUR BASIC PROGRAM

This section provides some techniques for optimizing BASIC programs; either making programs more efficient in regard to the time they need to execute, or in the amount of memory they require. Many of the techniques described here reduce execution time as well as the amount of memory used for 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 the elimination of extraneous program material. The keyword LET should be removed from any assignment statements, since it is not needed. Once the program is running correctly, REM statements may be removed since they take up memory space, and must be skipped over during program execution, thus increasing execution time. Variable names should be removed 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. These four bytes are "traded" for the statement separator," \", when two lines are compressed.

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 638Ø8, 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 constant such as 63488 is used heavily in the program, 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; and 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. 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 the heavily used sections are identified, optimization can then be accomplished with some confidence that it will make a positive 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 in finding "bugs."

Example: see next page

Example: (This example is similar to the sample program TIMER in Appendix B)

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 ! "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(0)
210 FOR I=1 TO 100
220 NEXT I
230 T=TIME(1)
240 !"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 !"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 !"A=B, FOR B=300, TAKES ABOUT",T/(100*60)," SECONDS."
>RUN
```

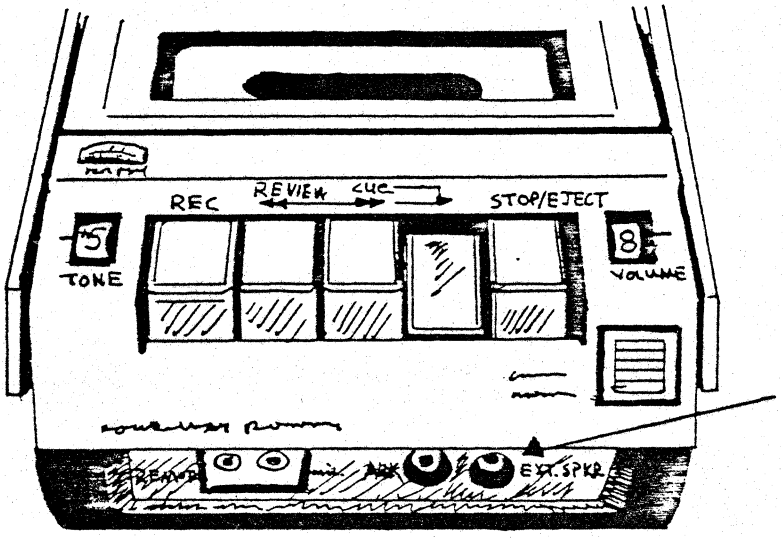
output

```
 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.
 >
```

Appendix A

LOADING BASIC, AND LOADING AND SAVING A BASIC PROGRAM

I. Using the Superscope C-103A Cassette Recorder



The cassette recorder is used to load BASIC and to save and load a BASIC program. The volume control should always be set at "8", and the tone control set at its highest setting, "+5". If the recorder is not powered by batteries, they should be removed. Whenever the recorder is used, the cable marked "MON" should be connected to the jack input labeled "ext. sp.".

The cassette recorder has five buttons marked:

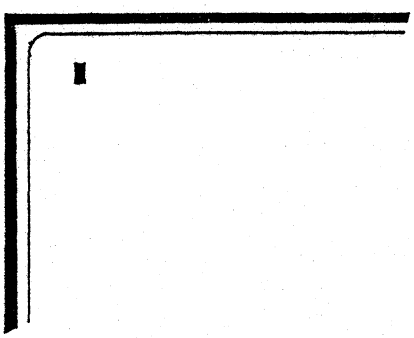
record:	used simultaneously with the normal speed cue button (▶) to record tapes.
review (◀◀):	used to rewind tapes.
cue (▶▶):	used to rapidly advance tapes.
cue (▶):	used to advance tapes at the normal play speed; it is the play button.
stop/eject:	used to stop tape or (when pushed in further) to eject tape.

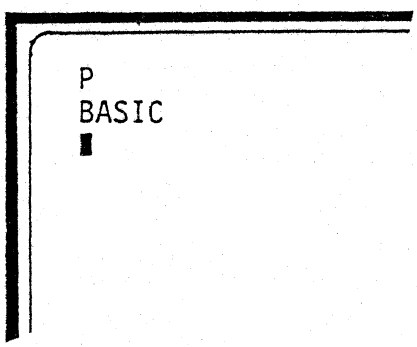
II. Loading BASIC from a Cassette Tape

A. Turn on the Poly 88 (or if your machine is already on, RESET by pressing the RESET button).

B. On the back of your Poly 88 machine is a switch marked "Poly/Byte". The position of this switch determines the mode of your machine; "Polyphase" or "Byte". If your BASIC tape is marked "Polyphase", make sure that this switch is in the "Poly" position; if your tape is marked "Byte", turn the switch to the "Byte" position.

C. The screen will appear blank except for a small white block at the upper left hand corner of the screen (the cursor).





D. Type:

PBASIC (to load BASIC written in "Polyphase" format), or
BBASIC (to load BASIC written in "Byte" format),

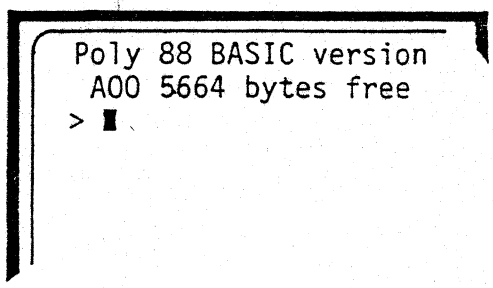
followed by carriage return.

E. Place cassette tape containing BASIC in cassette deck. Rewind tape. Then push normal speed cue button (▶).

F. A message will appear at the top of the monitor screen indicating which version of BASIC is being loaded (give it a few seconds to appear). As the tape is loaded, record numbers will appear on the screen along side the tape name. This will indicate that the tape is being loaded correctly. (For example, BASIC ØØØ1).

G. After the tape is loaded, BASIC will respond with a message at the top

of the screen, again identifying the BASIC version loaded, and giving the number of bytes available in memory.



H. A BASIC prompt will be printed on the screen indicating that BASIC has finished loading and is ready for your instructions.

Possible Problems

If a question mark appears instead of a record number when the tape is being loaded, the tape is not being correctly loaded. Several causes: volume control too low, interrupted tape, checksum error, damaged tape, etc. Try again with increased volume.

III. Saving and Loading a BASIC Program

A. Loading a BASIC Program

If you are loading a BASIC program from cassette tape, make sure that BASIC has already been loaded in your machine. Before loading a BASIC program, do not hit the reset button on your Poly 88 -- that will cause it to go to the monitor program. In order to execute BASIC programs, BASIC must already be loaded in your machine.

We will go through the process of loading a BASIC program using a program from Appendix B, Sample Programs. These programs have been included on the cassette tape labeled BASIC Sample Programs. We will assume that you

want to run the program named "ROSES".

1. Place the cassette tape labeled BASIC Sample Programs in the cassette recorder. Rewind the tape. This tape has been recorded in "Byte" format. Therefore turn the "Poly/Byte" switch in the back of your machine to "Byte". (note: a "Byte" tape may be loaded into your machine even if the BASIC you have loaded into the Poly 88 is recorded in "Polyphase").

2. Type:

   LOAD,ROSES,B (this program is loaded in "Byte" format. If the tape had been saved in "Polyphase" format, you would have typed LOAD,ROSES,P).

   Note: a program must be loaded in the same format ("Polyphase" or "Byte") in which it was saved, and with the same name it was saved under. This does not mean that the BASIC program must be in the same format as the BASIC that you have loaded into the machine. You may run "Byte" BASIC programs on "Polyphase" BASIC and vice versa, as long as the "Poly/Byte" switch in the back of the Poly 88 is in the appropriate position for the BASIC program that you are loading.

3. Make sure that the only cable connected to the cassette recorder is the cable labeled "MON" in the jack input labeled "ext. sp.".

   Type a carriage return. Depress the normal speed cue button (▶).

   BASIC will respond with the message "Working...."

   In the case of the example above, ROSES, you will see the name of another program appear on the screen (without record numbers) before you see ROSES appear. This is the program which is on the cassette tape before the program that you are asking for, ROSES.

BASIC skips over this program, but gives you its name, so that you know where you are on the cassette tape.

When BASIC reaches the program that you have asked for, the name of that program will appear along side its record numbers as they are loaded from the tape.

After all records of the program have been loaded, BASIC will display a prompt symbol, >, to indicate that it is ready for new instructions.

4. If at any time you decide that you wish to interrupt the process of loading a program, use of the Control-Y command will return you to BASIC. Use of the Control-Y command will erase anything in working memory and clear all variables and strings, so do not use it if you have anything on the screen you wish to save. To type a Control-Y command, hold down the CTRL key and type Y.

B. Running a Program Loaded from Cassette Tape

After a program has been loaded from tape, the program will either go into regular execution mode or auto-execute mode. If the program has been recorded in regular execution mode, it will not begin executing until you type "RUN" and a carriage return after a BASIC prompt; >. If the program has been saved in auto-execute mode, it will begin executing immediately after loading without further user input.

If, after having correctly loaded your program, BASIC responds without a prompt, >, you know that the program has not been saved in auto-execute mode and requires a "RUN" and a carriage return after a prompt in order to execute. To save your programs in auto-execute mode, see C, Saving a BASIC Program.

After you have finished with one program, and wish to load another, you must type "SCR" after a prompt. This will clear the old program from memory and ready the memory to receive the new program. You may have only one program at a time in working memory. Then follow the directions above, specifying the name of the program you wish to load. You may interrupt a program at any time by using the Control-Y command.

Example:

If you loaded ROSES, typed RUN and then a carriage return, ROSES would begin to run. You then might decide to interrupt its execution by use of the Control-Y command. After typing SCR, you would then be free to load another program. In the example, the user wanted to see ATAN, which is located before ROSES on the tape. After the user gives a Control-Y command, interrupting the ROSES program, the user must type LOAD,ATAN,B and a carriage return. Then the user must rewind the tape to the point at which ATAN begins on the tape. ATAN is the first program on the tape, so the tape should be rewound to the beginning, and then started forward again by pressing the (or play button on the recorder). Below is a recreation of what you might see on your screen during this whole process.

```
>
>LOAD,ROSES,B
 Working...

 ATAN
 ATAN
 ATAN
 ATAN
 ATAN
 ROSES ØØØ
 ROSES ØØ1
 ROSES ØØ2
 ROSES ØØ3
 ROSES ØØ4
 ROSES

>RUN
```

(Example continued on following page)

```
SAMPLE PROGRAM ROSES
I WILL PLOT THE EQUATION FOR A FAMILY OF ROSES BASED
ON THE STARTING NUMBER YOU GIVE ME (>2, PLEASE!).
STARTING N=(Control-Y command used here)

Interruption in line 310

>>SCR
>LOAD,ATAN,B
 Working

 ATAN 000
 ATAN 001
 ATAN 002
 ATAN 003
 ATAN
```

Possible Problems

"Checksum error" indicates that BASIC is unable to load the cassette tape program. This may be the restult of an attempt to load the program in the wrong format (for example, to load a "Byte" program with the "Poly/Byte" switch in the back of the machine turned to "Poly"). It may also be caused by tape damage, an interrupted tape, incorrect volume and tone control settings, a checksum error, etc.

D. Saving a BASIC Program

Once you have created a BASIC program, you may wish to record--or save--that program on tape.

1. To save a program, choose a name for your program that is less than 8 characters long. For example; name your program POETRY.
2. Attach the "BIPHASE" or "BYTE" cable to the jack input labeled "aux", depending upon the format you wish to use for recording your program ("Polyphase" or "Byte"). Remember to set the "Poly/Byte" switch on the back of

your Poly 88 to the proper format position.

3. Type:

   SAVE,POETRY,P (to save a program in "Polyphase" format)

   SAVE,POETRY,B (to save a program in "Byte" format)

   Do not hit carriage return.

4. Rewind the cassette tape, and push down the record and play button (▶) simultaneously.

5. When the tape leader disappears and the recording tape appears in the cassette deck window, hit carriage return.

6. BASIC will respond with the message "Working...", and give the record numbers of the current tape records as they are recorded on the cassette tape.

7. After the tape has been successfully recorded, BASIC will respond with a prompt,>, to indicate that it is ready for new instructions.

8. It is possible to save a program in auto-execute mode. If saved in this mode, the program will begin executing immediately after being loaded, without the use of the RUN command. To save a program in auto-execute form, use the standard form of the SAVE command (SAVE,NAME,P or B), but replace the first comma with a semicolon (SAVE;NAME,P or B).

E. Default Format for SAVE, LOAD and VERIFY

If P or B is not specified in the SAVE, LOAD, or VERIFY commands, the default format, "Byte is used by BASIC. (See Appendix D).

F. Verifying Your Saved BASIC Programs

Let us say that you have written a program named XANADU. You want to do something else with your Poly 88 now, so you decide to save the program on tape for future use. When you save a BASIC program on cassette tape, you don't actually transfer it to the cassette tape; it's merely copied onto the tape from memory. After you save the program on tape, you still have the original program in memory. You may wish to check the recorded version against the original program still in memory to make sure that the recording is good. BASIC provides a way for you to do this; the VERIFY command.

Be careful to use the VERIFY command before any changes are made to the program still in memory or before you LOAD another program (LOAD erases everything in working memory).

Type:

VERIFY,XANADU,P (if the program was saved in "Polyphase" format)

VERIFY,XANADU,B (if the program was saved in "Byte" format),

followed by a carriage return.

Make sure that the "MON" cable is the only cable attached to the recorder; once again check the "Poly/Byte" switch on the back of your machine to see if it is in the proper position for your tape.

With the rewound tape in position in the cassette deck, and the unmodified program still in memory, type the VERIFY command and depress the play button (▶) on the recorder.

If the program read from the tape matches the program in memory identically, the record names and numbers will appear on the screen as they would for a LOAD command. A prompt symbol should appear if the tape has been verified.

If at any point the program in the Poly 88's memory does not match the program read from the tape, a VERIFY error message will result. The tape will not verify if the program has not been saved correctly, if there is tape damage, or if the original program has been changed in memory since it was saved on tape.

G. Interrupting Loading or Saving a BASIC Program

A Control-Y command may be used to interrupt saving or loading a program. If used while saving a program, the program on the tape will probably contain invalid material; if used while loading a tape, the equivalent of a SCR command is executed, erasing any program lines, variable values, etc. in working space memory.

Appendix B

SAMPLE PROGRAMS

The cassette tape labeled BASIC SAMPLE PROGRAMS contains 10 programs which demonstrate some of the capabilities of the Poly 88 BASIC. These programs of varying complexity are provided in this manual so that the novice user can load these programs and see the programs in execution. The programs in this section were contributed either by R. T. Martin or S. Tytonida; the listings in this section of the manual were made from the files on the sample program tape. Where practical, a sample run of the program is included with the listing, although most of the programs rely on the use of the video display.

The sample program tape is recorded in "Byte" format. Some of the programs have been recorded to begin execution automatically, without further user input after having been loaded. Others require the user to type RUN after a prompt. To run one of the programs, follow the directions given in Appendix A for loading programs from cassette tape. Use one of the program names below.

The names of the 10 sample programs on the tape are:

ATAN
ROSES
ORBIT
PRIMES
RHIST
SORT
CLOCK
NEST
TIMER
FACT

Sample Program ATAN

This program was written to demonstrate the use of multi-line functions, as well as to provide an algorithm for computing the arctangent. The approximation utilized by this program is from Approximations for Digital Computers, by Cecil Hastings, Jr., Princeton University Press, 1955. ATAN is organized for clarity, not for computational speed. Note that in this program, as in all the sample programs, and any program that is intended for general use, that the user is informed as to what is desired by the program as input, and then that input is validated to some extent. This process of explanation and then validation is central to the difference between a random computer "program" and a program that is a product.

```
>LIST
100 REM SAMPLE PROGRAM "ATAN"
110 REM DEMONSTRATES MULTI-LINE FUNCTIONS,
120 REM AND GIVES AN EXPANSION FOR FINDING ARC-TANGENT OF
130 REM OF AN ANGLE IN RADIANS
140 !"SAMPLE PROGRAM ATAN"
150 !"GIVE ME A POSITIVE NUMBER, AND I WILL TELL YOU WHAT"
160 !"ANGLE IN RADIANS AND DEGREES IT IS THE TANGENT OF,"
170 !"TO 5 DIGITS OF PRECISION"
180 INPUT "NUMBER = ",X
190 IF X=>0 THEN 210
200 PRINT "MUST BE ZERO OR GREATER, PLEASE!"\GOTO 180
210 PRINT "THAT'S THE TANGENT OF",FNT(X)," RADIANS, OR",
220 PRINT 360*FNT(X)/(2*3.1415926)," DEGREES."
230 GOTO 180
240 REM FUNCTION FOR COMPUTING ARCTANGENT
250 REM SOURCE IS "APPROXIMATIONS FOR DIGITAL COMPUTERS"
260 REM BY CECIL HASTINGS, JR. PUBLISHED BY PRINCETON
270 REM UNIVERSITY PRESS, 1955.
280 DEF FNT(R)
290 S=(R-1)/(R+1) \REM CONVERT THE RANGE
300 T=0\T=.99997726*S-.33262347*(S^3)+.19354346*(S^5)
310 T=T-.11643287*(S^7)+.05265332*(S^9)-.01172120*(S^11)
320 RETURN 3.1415926/4+T
330 FNEND
340 REM NOTE THAT THE COMPUTATION IS NOT OPTIMIZED FOR SPEED,
350 REM BUT TO SHOW THE ALGORITHM AND THE CONSTANTS!
>RUN

SAMPLE PROGRAM ATAN
GIVE ME A POSITIVE NUMBER, AND I WILL TELL YOU WHAT
ANGLE IN RADIANS AND DEGREES IT IS THE TANGENT OF,
TO 5 DIGITS OF PRECISION
NUMBER = 1
THAT'S THE TANGENT OF .78539815 RADIANS, OR 44.999999 DEGREES.
NUMBER = 1.733
THAT'S THE TANGENT OF 1.0474356 RADIANS, OR 60.013641 DEGREES.
NUMBER =
Interrupted in line 180
>>
```

Sample Program ROSES

This program is a "number cruncher". A number cruncher is a program that does an extraordinary amount of computation. ROSES is such a program. For each point displayed on the screen, two sines and a cosine must be calculated (lines 35Ø-36Ø). 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 27Ø (1ØØ as recorded on the tape). 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 1ØØ (or even 1ØØØ), and observe the results. Try K = 5ØØ and starting N = 83. If you are mathematically inclined, examine the effect of sampling the rose equation in closed form. Why is it the case that for N > 1ØØØ we do not see a solid white screen (for K = 5ØØ), but instead see some very interesting patterns?

```
100 REM SAMPLE PROGRAM "ROSES"
110 REM THIS PROGRAM PLOTS ROSES ON THE VIDEO SCREEN.
120 REM THE GENERAL FORM OF THE ROSE, IN POLAR FORM, IS
130 REM R=A*SIN(N*T) WHERE A IS THE MAXIMAL RADIUS, AND
140 REM T IS THE ANGLE THETA, WHICH GOES FROM 0 TO 2*PI
150 REM RADIANS TO GENERATE THE ROSE. TO PLOT THIS FUNCTION
160 REM IN THE CARTESIAN COORDINATE SYSTEM, WE USE THE
170 REM TRANSFORMATIONS X=R*COS(T)+X1 AND Y=R*SIN(T)+Y1,
180 REM WHERE (X1,Y1) IS THE COORDINATES OF THE POINT WE
190 REM WISH TO CALL THE ORIGIN. THIS GIVES US THE EQUATIONS
200 REM X=63.5+44*SIN(N*T)*COS(T), Y=23.5+22*SIN(N*T)*SIN(T)
210 REM TO SPEED UP THE COMPUTATION, WE FACTOR OUT THE TERM
220 REM SIN(N*T) TO GIVE THE EQUATIONS SHOWN BELOW. NOTE
230 REM THAT WE ONLY COMPUTE K POINTS ALONG THE CURVE; THIS
240 REM GIVES US AN INTERESTING SAMPLING EFFECT FOR LARGE N.
250 REM WE INPUT A STARTING N, AND GENERATE ROSES FOR N
260 REM DECREMENTING DOWN TO 2.
270 K=100\REM CHANGE FOR MORE OR LESS POINTS
280 PRINT CHR$(12),"SAMPLE PROGRAM ROSES"
290 !"I WILL PLOT THE EQUATION FOR A FAMILY OF ROSES BASED"
300 !"ON THE STARTING NUMBER YOU GIVE ME (>2, PLEASE!)."
310 INPUT "STARTING N =",L
320 IF L<2 THEN !"...GREATER TNAH 2, PLEASE!"\GOTO 310
330 FOR N=L TO 2 STEP -1
340 PRINT CHR$(12),\PRINT "N =",N\PLOT 0,44,0
350 FOR T=0 TO 2*3.14159 STEP 2*3.14159/K
360 S=SIN(N*T)\X=63.5+44*S*COS(T)\Y=23.5+22*S*SIN(T)
370 PLOT X,Y,1\NEXT
380 NEXT \ GOTO 270
>RUN
```

Sample Program ORBIT

The ORBIT program simulates the motion of two massless particles in motion about a force center. To describe them as "massless" particles is another way of stating that they do not interact with one another. They interact only with the force center.

This program was run with the 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 2ØØ, 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 on 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 the Poly 88 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.

```
100 REM SAMPLE PROGRAM "ORBIT"
110 REM DEMONSTRATES PLOT FUNCTION IN DISPLAYING THE
120 REM ORBITS OF TWO MASSLESS PARTICLES ABOUT A FORCE CENTER
130 REM SIMPLE 2 BODY ORBITAL KINEMATICS PROGRAM
140 REM KINEMATICS EQUATIONS BY R. BALLARD, PROGRAMMING
150 REM BY R. MARTIN, BASIC UNDERSTANDING AND EXPLANATION
160 REM OF MOTION BY I. NEWTON
165 REM NOTE: ORGANIZED FOR SPEED, NOT EXECUTION!!!
170 PRINT CHR$(12), \ PLOT 0,47,0
180 PLOT 50,25,0\PRINT CHR$(128+14)\PLOT 0,21,0
190 X1=3\X2=0\V1=0\V2=.5\T=0\D=.1
200 D=.5\REM CHANGE D FOR MORE OR LESS ACCURACY IN ORBITS
210 X3=2\X4=0\V3=0\V4=-.6
220 PLOT H,V,0 \ H=10*(X1+5)\V=5*(X2+5)\PLOT H,V,1
230 PLOT H1,H2,0\H1=10*(X3+5)\H2=5*(X4+5)\PLOT H1,H2,1
240 X1=X1+V1*D\X2=X2+V2*D\X3=X3+V3*D\X4=X4+V4*D
250 S=X1*X1+X2*X2 \ R=SQRT(S)\S=D/(R*S)\V1=V1-S*X1\V2=V2-S*X2
260 S1=X3*X3+X4*X4\R1=SQRT(S1)\S1=D/(R1*S1)\V3=V3-S1*X3
270 V4=V4-S1*X4\T=T+D\GOTO 220
>REM DOES NOT DISPLAY WELL ON HYTYPE!!!!
>
>
>
```

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 5ØØ 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 manner which 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.

```
100 REM SAMPLE PROGRAM "PRIMES"
110 REM FIND AND PRINT PRIME NUMBERS.
120 REM MARCH 1977, S. TYTONIDA
130 REM THE LIST N IS USED TO HOLD THE FIRST 500 PRIMES-
140 REM IN TESTING TO SEE IF A NUMBER IS PRIME, WE ONLY NEED
150 REM TO LOOK FOR FACTORS THAT ARE LESS THAN OR EQUAL TO
160 REM THE NUMBER; IN FACT, WE ONLY NEED TO CHECK PRIME
170 REM FACTORS LESS THAN OR EQUAL TO THE SQUARE ROOT OF THE
180 REM NUMBER. RATHER THAN CALCULATE A SQUARE ROOT EVERY TIME,
190 REM WE INSTEAD KEEP A POINTER, L, INTO THE LIST OF PAST
200 REM PRIMES, AND BUMP THAT UP AS NEEDED. NOTE THAT WE ONLY
210 REM TEST ODD NUMBERS. THE NUMBER WE DISPLAY IN THE MIDDLE
220 REM OF THE SCREEN IS THE LATEST PRIME, THE NUMBER AT THE
230 REM BOTTOM IS THE CURRENT TEST BOUND. THE RATHER
240 REM BAROQUE EXPRESSION (INT(M/N(P))*N(P)-M) GIVES THE
250 REMAINDER OF DIVIDING THE NUMBER M BY PRIME FACTOR N(P).
260 REM IF THE REMAINDER IS ZERO, THE NUMBER CANNOT BE PRIME.
270 REM IF NON-ZERO, WE MUST TEST PRIME FACTORS THRU N(L).
280 REM IF NONE OF THOSE ARE DIVISORS, WE HAVE A NEW PRIME,
290 REM AND IF K<500, WE STUFF IT ONTO THE LIST. MY THANKS
300 REM TO IRA BAXTER FOR EXPLAINING TO ME, MANY MOONS AGO,
310 REM WHY YOU DON'T NEED TO CALCULATE SQUARE ROOTS EVERY
320 REM TIME, AND TO THE ANCIENT GREEKS THAT DISCOVERED THE
330 REM MAGIC AND MADNESS OF PRIME NUMBERS.
340 REM REMEMBER: (2^19937)-1 IS PRIME!
350 DIM N(500)
360 PRINT CHR$(12),\PLOT 0,47,0\REM CLEAR SCREEN AND ERASE CURSOR
370 N(1)=2\ N(2)=3\ N(3)=5
380 K=2\L=2\M=5
390 P=1\IF M>N(L)^2 THEN L=L+1\GOTO 390
400 IF (INT(M/N(P))*N(P)-M)=0 THEN M=M+2\GOTO 390
410 IF P=>L THEN 420 ELSE P=P+1\GOTO 400
420 K=K+1\IF K<500 THEN N(K)=M
430 PLOT 55,23,0\PRINT M," IS PRIME!"\PLOT 0,20,0\M=M+2\GOTO 390
>
>
>
```

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 190 to 230. 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 270 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, pseudo-random numbers, and methods of testing and generating them. The random number generator used in BASIC was provided by Eric Rawson.

```
100 REM SAMPLE PROGRAM "RHIST"
110 REM USES THE PLOT FUNCTION AND PRODUCES A HISTOGRAM
120 REM SHOWING THE DISTRIBUTION OF THE RANDOM NUMBER
130 REM GENERATOR, AND PERCENTAGE DISTRIBUTIONS
140 DIM A(100),Y(100),Q(10)
150 PRINT CHR$(12),\PLOT 0,47,0\REM CLEAR THE SCREEN
160 N=100 \ S=100 \ REM N IS THE SAMPLE SIZE, S IS TOTAL SAMPLES
170 FOR I=1 TO 100\Y(I)=7\NEXT\REM INITIALIZE HISTO BARS
180 PLOT 121,43,0\PRINT "%%%"\PLOT 0,40,0\REM PRINT DIST. HEADER
190 FOR I=1 TO N\K=INT(100*RND(0))+1\A(K)=A(K)+1\NEXT
200 H=-3\M=0\ REM H IS HIGHEST # SEEN, M=SUM
210 REM COMPUTE SUM (AREA UNDER CURVE) AND FIND HIGH VALUE
220 FOR I=1 TO N\M=M+A(I)\IF A(I)>H THEN H=A(I)
230 NEXT
240 F=.1\G=0\J=1\REM PUT UP DECILE (%%%) POINTS
250 FOR I=1 TO N\G=G+A(I)\IF G<F*M THEN 290
260 IF Q(J)=I THEN 280\REM THE VALUE HAS NOT CHANGED
270 PLOT 118,3*J+10,0\PRINT I\PLOT 0,3*J+7,0\REM PRINT POINT
280 Q(J)=I\J=J+1\F=F+.1
290 NEXT
300 PLOT 0,3,0\PRINT "N =",S," MAX =",H\PLOT 0,0,0
310 REM NOW PLOT BARS. NOTE THAT WE SCALE, SO THAT THE
320 REM LARGEST BAR IS 39 HIGH. X=2+I+INT((I-1)/10)
330 REM GENERATES A BLANK SPOT EVERY 10 TO AID IN COUNTING
340 REM THE BARS ON THE SCREEN.
350 REM WE SEE IF A BAR HAS CHANGED, HAS GROWN, OR WHAT, AND
360 REM DO THE RIGHT THING FOR EACH CASE TO OPTIMIZE OUR DRAWING.
370 FOR I=1 TO 100\V=7+INT(39*A(I)/H)\X=2+I+INT((I-1)/10)
380 IF V=Y(I) THEN 420
390 IF V<Y(I) THEN 410
400 FOR J=Y(I) TO V\PLOT X,J,1\NEXT\GOTO 420
410 FOR J=Y(I) TO V STEP -1\PLOT X,J,0\NEXT
420 Y(I)=V\NEXT
430 S=S+N\GOTO 190
>REM ANOTHER PROGRAM THAT DOES NOT DO WELL ON THE HYTYPE....
>
>
>
>
>
>
>
```

Sample Program SORT

Sort was written to demonstrate two differing methods of sorting, and the relative efficiencies involved in each. Sort also demonstrates the utility of a small, personal computer with the right balance of software features in computer science education. One of the authors (Martin) feels he learned more about sorting algorithms and algorithmic analysis by sitting down with Vol. III of Knuth and the Poly 88, and building sorting algorithms and testing them than he did in one three-month academic quarter of formal classes.

This program also demonstrates the use of PEEK and POKE for examining and modifying memory locations, especially the video board 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 Doanld Knuth, which is devoted entirely to sorting and serching, rather than volumes I or II.

```
100 REM SAMPLE PROGRAM "SORT"
110 REM THIS PROGRAM USES THE PEEK AND POKE FUNCTIONS TO
120 REM MANIPULATE THE CONTENTS OF THE VIDEO BOARD, AND
130 REM MORE IMPORTANT, DEMONSTRATES TWO TECHNIQUES OF
140 REM SORTING INFORMATION: THE VENERABLE BUBBLE SORT,
150 REM AND THE SIMPLE, BUT VASTLY SUPERIOR "SHELL" SORT.
160 Z=RND(TIME(1)/65536)\REM RANDOMIZE....
170 DIM P(256)\REM HOLDS STUFF TO SORT
180 DIM H(10)\ REM HOLDS INCREMENTS USED BY SHELL SORT
190 REM CALCULATE INCREMENTS FOR SHELL SORT ALGORITHM
200 H=4\FOR I=1 TO 10\H(I)=H\H=3*H+1\NEXT
210 GOSUB 410\REM GENERATE LIST OF STUFF TO SORT
220 PRINT CHR$(12),\INPUT "HOW MANY THINGS TO SORT (2-256)?",N
230 IF (N>256) OR (N<2) THEN 220\ REM FILTER ANSWER
240 PRINT "WHICH SORT DO YOU WANT TO USE:"
250 PRINT "      1 BUBBLE SORT"
260 PRINT "      2 SHELL SORT"
270 INPUT"1 FOR BUBBLE, 2 FOR SHELL : ",M
280 IF (M<>1) AND (M<>2) THEN 270\REM FILTER ANSWER
290 INPUT "DO YOU WANT THE SAME TEST PATTERN (Y OR N)?",A$
300 IF A$="N" THEN GOSUB 410\GOTO 320
310 IF A$<>"Y" THEN 290
320 O=63487\ REM SCREEN ORIGIN (F800 HEX) -1
330 PRINT CHR$(12),\PLOT 0,47,0\REM CLEAR THE SCREEN
340 FOR I=1 TO N\POKE I+O,P(I)\NEXT\REM FILL SCREEN WITH CRUD
350 S=TIME(0)\W=0\REM TIME AND NUMBER OF SWAPS
360 ON M GOTO 440,520
370 PLOT 0,12,0\PRINT "SORTED ",N," THINGS IN",W," SWAPS, AND",
380 PRINT TIME(1)/60," SECONDS."
385 INPUT "TRY AGAIN (Y OR N)?",A$\IF A$="Y" THEN 220
390 IF A$<>"N" THEN 385
400 STOP\GOTO 220\REM GOTO SO THAT 'CON' WILL CONTINUE PROGRAM.
410 REM GENERATE NEW PATTERN IN P
420 PRINT "THINKING...."
430 FOR I=1 TO 256\P(I)=128+127*RND(0)\NEXT\RETURN
440 REM BUBBLE SORT. WE WANDER DOWN THE LIST, LOOKING FOR
450 REM TWO ELEMENTS OUT OF ORDER, AND SWAP 'EM WHEN WE FIND EM.
460 S=TIME(0)
470 K=N
480 F=0\FOR I=O+1 TO O+K-1
490 L=PEEK(I)\M=PEEK(I+1)\IF L<=M THEN 510
500 F=1\POKE I+1,L\POKE I,M\W=W+1
510 NEXT\K=K-1\IF F=0 THEN 370 ELSE 480
520 REM SHELL SORT. THIS IS FROM KNUTH VOLUME 3, ALGORITHM D.
530 S=TIME(0)\W=0
540 FOR Q=1 TO 9\IF H(Q+1)>N THEN EXIT 560
550 NEXT
560 FOR J=Q TO 1 STEP -1
570 F=0\H=H(J)\FOR I=O+1 TO O+N-H
580 L=PEEK(I)\M=PEEK(I+H)\IF L<=M THEN 600
590 F=1\POKE I,M\POKE I+H,L\W=W+1
600 NEXT\IF F>0 THEN 570
610 NEXT\GOTO 470\REM FINISH WITH BUBBLE
>
>
>
```

Sample Program CLOCK

This program demonstrates the real-time clock function available in BASIC. It also uses formatted print in displaying the time (lines 26Ø and 4ØØ), PEEK, POKE, and OUT. Without redevelopment, CLOCK turns the POLY 88 into a very expensive, and inaccurate clock. After the program was written, it was determined that it was not very accurate, loosing two or three minutes an hour. Solve the problem of this inaccuracy, and in so doing you will learn about utilization of the time function. It is also a simple matter to modify the program to display every second.

```
¬
>
>LIST
100 REM SAMPLE PROGRAM "CLOCK"
110 REM THIS PROGRAM DEMONSTRATES THE USE OF THE REAL TIME
120 REM CLOCK AVAILABLE THROUGH THE BASIC "TIME" FUNCTION
130 REM IF YOU HAVE AN AI CYBERNETICS MODEL 1000 SPEECH
140 REM SYNTHESIZER AT OUTPUT PORT 254, IT WILL GENERATE
150 REM "TICK-TOCK" NOISES....
160 REM WRITTEN MARCH 1977 S. TYTONIDA
170 PRINT CHR$(12),"SAMPLE PROGRAM CLOCK"
180 PRINT "AFTER YOU GIVE ME THE CURRENT TIME IN HOURS AND"
190 PRINT "MINUTES, I WILL BE A CLOCK!"
200 INPUT "WHAT HOUR IS IT (0-23)?",H
210 H=INT(H)\IF (H<0) OR (H>23) THEN 200
220 INPUT "WHAT MINUTE DO I START WITH (0-59)?",M
230 M=INT(M)\IF (M<0) OR (M>59) THEN 220
240 S=0 \ REM SECONDS COUNTER
250 PRINT " WHEN YOU HIT RETURN, I WILL START BEING A CLOCK AT"
260 PRINT %2I,H,":",M,":",0," O'CLOCK",
270 INPUT "(HIT RETURN TO START)",A$
280 PRINT CHR$(12),\PLOT 0,47,0
290 K=43 \ REM 'TICK' FOR AI CYBERNETICS BOARD
300 W=220\ REM SYMBOL FOR THE CLOCK
310 O=63488+32+8*64\ REM IN THE MIDDLE OF THE SCREEN
320 Z=TIME(0)
330 IF TIME(1)<60 THEN 330
340 IF K=43 THEN K=47 ELSE K=43
350 IF W=220 THEN W=175 ELSE W=220
360 OUT 254,K\POKE O,W\OUT 254,0
370 S=S+1\IF S<>60 THEN 320 ELSE S=0
380 M=M+1\IF M<>60 THEN 400
390 M=0\H=H+1\IF H=24 THEN H=0
400 PLOT 0,47,0\PRINT %2I,H,":",M,":",S\PLOT 0,43,0\GOTO 320
>
>
>REM NOT VERY INTERESTING ON A HYTYPE!!!
>
>
>
>
>
```

Sample Program NEST

This is a very bizarre program. It was thought up and written while preparing this manual. The question came up, "Well, just how many FOR-NEXT's can you nest in a 16K machine?" This program provides the answer. Basically, it uses the OUT Ø feature of BASIC that allows characters to be put in BASIC's input buffer to write a program. The function on lines 23Ø to 26Ø, when called with a string argument, places this string followed by a carriage return into the input buffer. The problem with having a program add statements to itself is that once the new statement is entered, execution of the program may not be continued: it must be completely restarted. For this reason we must devise some means of keeping track of our progress in the task of adding statements to the program. On each iteration through the process, we need to generate a FOR statement, and its accompanying NEXT, and then the command RUN to start the process over. We keep track of the line number we generated in the variable L, the letter of the alphabet we are generating FOR statements with in I, and the number following the variable in the variable J. The key to the process may be seen in line 15Ø; in this line we produce a NEW line 11Ø, with the updated values for L, I, and J. In this manner we can retain some memory of the program's last "life" in its new incarnation. Lines 1ØØ through 18Ø generate a new line 11Ø, the FOR and NEXT statements, and the RUN command in the input buffer, and then the program stops. When this happens, BASIC reads from its buffer, gobbling up the characters we have placed there. When we generate the desired number of FOR-NEXT pairs, controlled by the check on I in line 14Ø, we go to the second part of the program, starting at line 19Ø. It is the purpose of this part of the program to DELETE the first part of the program, delete itself, generate a PRINT statement at line 5ØØØ, and then run the constructed program, which consists of FOR-NEXT statements, and one PRINT. If you run this program and examine the line number on the last FOR statement, you can get the answer to the question, "How many FOR-NEXT loops can we nest?"

```
>LIST
100 DIM S$(50),A$(11),B$(26)
110L= 1036\I= 4\J= 4
120 A$=" 0123456789"\B$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
130 L=L+1\J=J+1\IF J=12 THEN J=1\I=I+1
140 IF I=12 THEN 190
150 Z=FNS("110L="+STR$(L)+"\I="+STR$(I)+"\J="+STR$(J))
160 Z=FNS(STR$(L)+"FOR "+B$(I,I)+A$(J,J)+"=1 TO 1")
170 L=9999-L\Z=FNS(STR$(L)+"NEXT "+B$(I,I)+A$(J,J))
180 Z=FNS("RUN")\STOP
190 Z=FNS("110GOTO200")\FOR I=120 TO 170 STEP 10\Z=FNS(STR$(I))
195 NEXT\GOTO 180
200 Z=FNS("100")+FNS("200")+FNS("190")+FNS("210")+FNS("220")
210 Z=FNS("5000!"+CHR$(34)+"!"+CHR$(34))+FNS("230")+FNS("260")
220 Z=FNS("180")+FNS("110")+FNS("240")+FNS("250")\GOTO 180
230 DEF FNS(S$)
240 S$=S$+CHR$(13)
250 FOR S1=1 TO LEN(S$)\OUT 0,ASC(S$(S1,S1))\NEXT\RETURN 0
260 FNEND
>
>
>REM WARNING: CLOSE EXAMINATION OF THIS PROGRAM MAY BE
>REM HAZARDOUS TO YOUR MENTAL STATE! (S. TYTONIDA)
>
>
```

Sample Program TIMER

This program was included to allow the user to time statements (as described in section 1Ø of this manual), to demonstrate the use of the TIME function, and to show that saying NEXT I is indeed slower in resulting program exectuion than saying simple NEXT. Because even the relatively slow 8Ø8Ø processor, and BASIC can execute statements much faster than the 6Ø 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 1ØØ samples, and it should be clear why we would want to use a larger number, say 1ØØØ, for the number of operations to time. In the timer program shown, how accurate, and repeatable are the results? If averaging over 1ØØØ samples is better than 1ØØ, wouldn't one million samples be better? How much better?

* Overhead time is time taken up by accomplishing things other than that which want to time.

```
>
>LIST
10 REM SAMPLE PROGRAM TIMER
20 REM THIS PROGRAM ALSO APPEARS AT THE END OF SECTION 10 OF
30 REM THE BASIC MANUAL. (S. TYTONIDA, MARCH 1977)
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 OF FOR-NEXT
170 !"LOOP OVERHEAD IS ABOUT",T/(100*60)," SEC PER ITERATION"
180 T1=T \ REM SAVE THAT OVERHEAD NUMBER
190 REM NOW TIME OVERHEAD WHEN WE USE "NEXT I"
200 T=TIME(0)
210 FOR I=1 TO 100
220 NEXT I
230 T=TIME(1)
240 !"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 LOOP OVERHEAD
310 !"A=300 TAKES ABOUT",T/(100*60)," SECONDS TO DO."
320 REM NOW SET B=300 AND TIME A=B
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 OVERHEAD
390 !"A=B, FOR B=300, TAKES ABOUT",T/(100*60)," SECONDS."
>RUN

LOOP OVERHEAD IS ABOUT .002 SEC PER ITERATION
VERSUS 2.6666667E-03 SEC PER ITERATION FOR NEXT I
A=300 TAKES ABOUT 3.1666667E-03 SECONDS TO DO.
A=B, FOR B=300, TAKES ABOUT 2.6666667E-03 SECONDS.
>
>REM YOU CAN INSERT YOUR FAVORITE EXPRESSION IN LINE 360,
>REM AND SEE HOW LONG IT TAKES TO EXECUTE....BON APETIT...
>
>
>
>
```

Sample Program FACT

FACT demonstrates multi-line finctions. The definition for the factorial function occurs on lines 26Ø to 28Ø. What happens when we call the function with the argument 1? With 2? With an argument greater than one, the function calls itself, saying, in effect; "I can return the factorial of three, if you give me the factorial of two". For an arbitrary number, this calling itself, or recursion, continues until the function is called with 1 as the argument, in which case it returns 1 to whomever called it,etc.

The notion of building the solution to a large problem by finding the solution to a simpler one is a very important idea in the use of computers. In fact, the idea of recursion is, to some extent, a more powerful tool in problem solving than the idea of loops, or iteration. With it we can build solutions to larger problems by building programs that break the problem down into smaller pieces that are easier to solve. But why the 17? The 17 appears because BASIC is not very efficient at accomplishing recursive functions, and one internal element of BASIC, called the "control stack",is rather small. With numbers larger than 17? Why don't you change line 2ØØ of the program and find out?

```
>
>LIST
100 REM SAMPLE PROGRAM "FACT"
110 REM THIS PROGRAM DEMONSTRATES RECURSIVE USE OF
120 REM MULTILINE FUNCTIONS IN FINDING FACTORIALS FOR
130 REM SMALL INTEGERS. (S. TYTONIDA, MARCH 1977)
140 !"SAMPLE PROGRAM FACT"
150 !"GIVE ME AN INTEGER SMALLER THAN 17, AND I WILL"
160 !"TELL YOU ITS FACTORIAL."
170 !"(TYPE CONTROL-Y TO STOP)"
180 INPUT "NUMBER IS ? ",X
190 IF (X-INT(X))<>0 THEN 150 \ REM NOT AN INTEGER
200 IF X>16 THEN 150 \ REM TOO BIG
210 IF X<0 THEN !X," FACTORIAL IS UNDEFINED!"\GOTO 180
220 !X," FACTORIAL IS",FNN(X)\GOTO 180
230 REM DEFINITION OF FACTORIAL. NOTE THAT THE FUNCTION
240 REM CALLS ITSELF. THIS IS AN EXAMPLE OF A RECURSIVE
250 REM FUNCTION. WE LIMIT TO <17 BECAUSE OF STACK SIZE...
260 DEF FNN(N)
270 IF N<2 THEN RETURN 1 ELSE RETURN N*FNN(N-1)
280 FNEND
>RUN

SAMPLE PROGRAM FACT
GIVE ME AN INTEGER SMALLER THAN 17, AND I WILL
TELL YOU ITS FACTORIAL.
(TYPE CONTROL-Y TO STOP)
NUMBER IS ? 7
 7 FACTORIAL IS 5040
NUMBER IS ? -3
 -3 FACTORIAL IS UNDEFINED!
NUMBER IS ? 2.2
GIVE ME AN INTEGER SMALLER THAN 17, AND I WILL
TELL YOU ITS FACTORIAL.
(TYPE CONTROL-Y TO STOP)
NUMBER IS ? 9
 9 FACTORIAL IS 362880
NUMBER IS ?
Interrupted in line 180
>>
>>
>>
>>
>>
```

Appendix C

THE BASIC CHARACTER SET

All characters and symbols in BASIC are stored in the machine as numbers (the ASCII code). The following list contains all of the characters in BASIC and their ASCII code in decimal representation. To print any character, type PRINT CHR$(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
             ! ! !

>
```

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

Numbers and Letters of the Alphabet

Ø	--	48	V	--	86
1	--	49	W	--	87
2	--	50	X	--	88
3	--	51	Y	--	89
4	--	52	Z	--	9Ø
5	--	53	a	--	97
6	--	54	b	--	98
7	--	55	c	--	99
8	--	56	d	--	1ØØ
9	--	57	e	--	1Ø1
A	--	65	f	--	1Ø2
B	--	66	g	--	1Ø3
C	--	67	h	--	1Ø4
D	--	68	i	--	1Ø5
E	--	69	j	--	1Ø6
F	--	7Ø	k	--	1Ø7
G	--	71	l	--	1Ø8
H	--	72	m	--	1Ø9
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	--	8Ø	u	--	117
Q	--	81	v	--	118
R	--	82	w	--	119
S	--	83	x	--	12Ø
T	--	84	y	--	121
U	--	85	z	--	122

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			

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 D

8080 MACHINE LANGUAGE INTERFACE

This section is written for those who understand 8080 machine language and wish to interface assembly language programs with Poly 88 BASIC. It will also be of help to those who wish to change the defaults for certain features in Poly 88 BASIC. For both these purposes, an understanding of the Poly 88 front panel mode of operation, for examining and modifying memory locations is assumed.

D.1 Default modes and flags

The following items are default values present in Poly 88 BASIC version A00 at the (Hexadecimal) locations shown:

Location	Contents	Description
2006	1A	Character code that when detected, causes entry to the Poly 88 front panel. The default as shown is a control-Z. This byte may be changed to another ASCII character code to change the front panel entry code, or to 00 to disallow entry to the front panel from BASIC.
2007	19	Interrupt character code for BASIC. Default is control-Y.
2008	42	Default mode for writing cassette tapes. The default is the character code "B," for byte. This may be changed to 50 (ASCII "P") to make the default mode Polyphase. Any other contents of this location will result in a syntax or other error when the default format is used in a tape command.
2009	3E	This is the ASCII character used by BASIC as the prompt. If this byte is changed to 00, BASIC will not prompt the user at the line entry or program continuation level.
205D-E	FF 49	Address 49FF is the end of BASIC.
2060-1	FF 4F	Address 4FFF is the starting address used in searching for the end of memory.

D.2 Changing memory limits, installing assembly language routines

An example of the proper method for installing assembly language interfaces to BASIC is given in the documentation for BPRINT, the printer driver for Poly 88 BASIC. The assembly language program should be written to load at address

4AØØ, past the end of BASIC. The program, in its initialization section, should modify locations 2Ø5D-E, and 2Ø6Ø-1 in BASIC, to set up memory limits. Locations 2Ø5D-E should be set to point after the end of the assembly language routine and any of its resident data. The address stored in 2Ø5D-E will be used as the beginning of BASIC data and program storage. If this address is above 4FFF, location 2Ø6Ø-1 must be changed to one plus the contents of 2Ø5D-E, the beginning location used in scanning for the end of memory that BASIC will use. In this manner, the assembly language routine modifies BASIC in such a way that it exists immediately following BASIC, and before BASIC program and data storage.

D.3 CALL interface

The CALL function is used to invoke assembly language routine. The format is either CALL (addr,val) or CALL (addr) where both addr and val are expressions that must evaluate to Ø<=addr<=65535 and Ø<=val, <=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 present in register pair HL will be converted to an integer and passed to the BASIC program as the value of the call.

D.4 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 anamalous program behavior, possibly resulting in the loss of the program and/or its data.

The PEEK function takes the form PEEK addr, val where addr is an expression evaluating to the range Ø<=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-repetitive results.

The POKE statement takes the form POKE addr, val where addr is an expression evaluating to the range Ø<=addr<=65535 for the memory address to modify, and Ø<=val<=255 for the 8 bit quantity to store at that address. As noted above, caution should be exercised in the use of the POKE statement.

D.5 8Ø8Ø IN and OUT

8Ø8Ø IN and OUT functions may be performed through BASIC using the INP function and the OUT statement, respectively. The format of the INP function is INP (port), where Ø<=port<=255 is the port address. INP(port) returns as an integer the 8 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 Ø<=port<=255 is the 8Ø8Ø port address as in INP above, and val is the 8 bit value Ø<=val<=255 that is sent to the specified post. 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 bahaviour, possibly resulting in the loss of the program and/or its data.

D.6 INP(Ø), INP(1), INP(2), and OUT Ø

The calls to INP with port addresses Ø-2 return data regarding the type-ahead.. INP(Ø) returns the status of the type-ahead buffer; Ø if the buffer is empty, and ≠ 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 the attempt to place characters into the input buffer when it is full will be ignored. Printing a control-X character will flush the input type-ahead buffer.

D. 7 Re-entering BASIC from Front Panel Display

To reenter BASIC from the front panel display, type: SPJ2ØØØ for "cold start" (BASIC assumes there is no program in effect); type SPJ2ØØ3 for "warm start" (BASIC assumes there is a program in the machine); and type SPJ49CØ to "warm start" from "B-print" (Printer Driver".) Then type carriage return and "G" to return to BASIC. 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$)
      190 FOR I=1 TO LEN(S$)\C=ASC(S$(I,I))\OUT 0,C\NEXT
      200 OUT 0,13\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$)
      190 FOR I=1 TO LEN(S$)\C=ASC(S$(I,I))\OUT 0,C\NEXT
      200 OUT 0,13\RETURN 0
      210 FNEND
      >>SCR
      >LIST
      >
      >
```

Appendix E: COMMANDS, FUNCTIONS AND KEYWORDS RECOGNIZED BY BASIC.

Next to each entry are the page numbers that refer to the manual location where information about the item may be found.

INTRINSIC FUNCTIONS, 52

INDEX