DATAPOINT DOS BASICPLUS BASICPLS

User's Guide

Version 2

April, 1980

Document No. 50335

DATAPOINT

PREFACE

BASICPLUS is an interactive, interpretive processor for the BASIC language.

BASICPLUS is available for two types of processors in the Datapoint product line. The first runs on any 5500 instruction set processor with at least 36K of memory, and requires any DOS version 2.4 or higher. The other version executes only on the 1500 processor with the 64K memory option. This version requires DOS.H version 2.5 or higher. BASICPLUS is structurally similar to and upwards compatible from DOSBASIC for the Datapoint 2200 DOS and Datapoint Diskette 1100. It is also upwards compatible from the 5500 DOS BASIC, BASIC55.

NOTICE

Datapoint strongly recommends that its customers use Datapoint Customer supplies. These disks, diskettes, cassettes, ribbons and other products are certified by Datapoint to meet all Datapoint Hardware specifications for consistent optimum performance.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION TO DOS BASICPLUS

1.1 History and Purpose of BASIC

BASIC, Beginner's All-purpose Symbolic Instruction Code, was initially developed at Dartmouth College in 1963-64 under the direction of Professors John G. Kemeney and Thomas E. Kurtz, with partial support from a National Science Foundation grant.

BASIC is distinguished from other programming languages in its concern for the unsophisticated or novice user. While BASIC is a general-purpose programming language, it is designed primarily to be easy to learn, easy to use and easy to remember.

BASIC programs are meant to be transparent; that is, they should do what would be expected of them on the basis of examination. Only a little knowledge of BASIC is needed to solve simple problems. Default conventions handle programming details for the beginner, enabling one to print before learning about formatting, and to use subscripted variables before learning about dimensioning. More advanced features of the language allow programmers to accomplish more sophisticated tasks and to override defaults.

1.2 Datapoint BASICPLUS

Datapoint BASICPLUS is an enhancement of (and is upwards compatible from) previous Datapoint BASICs: DOSBASIC for the Datapoint 2200 DOS and Datapoint Diskette 1100, as well as the Datapoint 5500 version of DOS BASIC, BASIC55.

Since an interactive environment facilitates learning, BASICPLUS is fully interactive. Statements can be entered directly to BASICPLUS and be individually checked for syntactic correctness. Any syntax errors are reported to the user immediately, thereby assisting both the programming and the learning process.

BASICPLUS can GET program text from (and SAVE program text to) disk files. While BASICPLUS has some editing facilities, the GET/SAVE feature enables the user to create or modify programs under EDIT and also to transfer BASIC programs from other systems.

1.2.1 Relationship to the Proposed National Standard

In the period from 1964 to 1974, many computer manufacturers and educational institutions produced implementations of BASIC, each differing slightly from the other. In January, 1974, the American National Standards Institute convened a subcommittee (X3J2) to produce a standard for BASIC. By 1978, the subcommittee produced a proposed standard, BSR X3.60 -- Programming Language Minimal BASIC. This proposed standard has not yet been ratified by ANSI. The subcommittee has continued its work, and is developing a substantially more comprehensive standard for BASIC at this time.

Some of the new features of BASICPLUS reflect the proposed standard; others are enhancements beyond the features of ANS Minimal BASIC.

After the Proposed National Standard is ratified, Datapoint intends to revise BASICPLUS to be in conformance with ANS Minimal BASIC. Programs written in BASICPLUS (or any of its predecessors) may require modification in order to be run on such a later version.

1.2.2 Features

The features of BASICPLUS provide a complete integration of Datapoint BASIC with all other Datapoint systems and offer the user not only an attractive independent BASIC system, but a powerful additional tool to enhance existing Datapoint systems. These features include:

* Numeric operations (+, -, *, /, ^, MIN, MAX)

* Numeric functions (SQR, INT, RND, ABS, SGN)

* Transcendental functions (EXP, LOG, SIN, COS, TAN, ATN)

* Relational operators (<, =, >, <=, >=, <>, #)

* Logical constants and operators (TRUE, FALSE, AND, OR, NOT)

* Chaining and keyboard-controlled execution

* Disk input/output in standard Datapoint file format

* Local, remote, or servo printer support configured automatically or by specification

* Printer column width configured automatically or by specification

* Unused printer support area released and added to user's work area

* Desk calculator execution of expressions

* Complete error messages and built-in debug aids

* Workspace save and restore on disk in compatable Datapoint file format

* Built-in program editor

* Long variable names to improve readability

* Multiple statements per line

* String and string array processing

* Co-ordinate micro positioning of servo printer for plotting

1.2.3 Differences from Previous Datapoint BASICs

BASICPLUS is upwards compatible from DOSBASIC and BASIC55 with a few exceptions.

1.2.3.1 Multiple-Statement Delimiter

Although in most cases, semicolons were interchangeable with commas (e.g. READ A;B) in previous versions of BASIC, BASICPLUS uses the semicolon as a multiple-statement separator; therefore, existing programs which use semicolon in place of comma must be modified before they will run on BASICPLUS. This change was necessary in order to make full use of the multiple-statement delimiter.

The PRINTs in a multiple-statement line must not use the comma as the multiple-statement delimiter. For example:

```
PRINT I,NEXT I
```

must be changed to:

```
        PRINT I,;NEXT I
```

where the comma causes the PRINT statement to use zoned printing, and the semicolon is the multiple-statement separator. The semicolon has no effect on the PRINT statement except to separate it from its successor. Statements such as:

```
        PRINT I;NEXT I
```

must be changed to:

```
        PRINT I;;NEXT I
```

if the user wishes to use unzoned output. Again, the first semicolon causes the PRINT statement to use compressed printing and the second semicolon is the multiple-statement character.

It is important to note that:

```
        PRINT I;NEXT I

             and

        PRINT I;;NEXT I
```

are not equivalent in BASICPLUS, in that the first example will output a single value per line, while the second will print multiple values per line.

1.2.3.2 The TAB Function

In previous versions of Datapoint BASIC, the PRINT statement's TAB function was defined in terms of "offset from leftmost print position" (i.e. 0-79). Because the standard BASIC print statement's TAB function is defined in terms of "addressing the n-th print position" (i.e. 1-80), BASICPLUS has been changed to implement tabbing in this fashion. The HP function can be used to position the cursor in the "offset" manner, and also to tab backwards as previous versions of DOS BASIC allowed (see chapter 11). In addition, previous versions of BASIC allowed any type of separator after the TAB function (e.g. PRINT "HELLO";TAB 10;"THERE"). BASICPLUS requires that a comma follow the function. The last example must be changed to: PRINT "HELLO";TAB 10,"THERE".

1.2.3.3 Processor and Memory

BASICPLUS is available for two different Datapoint processor types. One version requires a 5500 instruction set processor with at least 36K of memory. If more memory is available, then the user's work space and number of dictionary entries will be noticeably increased. The other version executes only on the Datapoint 1500, and requires the 64K memory option. It will run with or without concurrent jobs active. The user's workspace will be about 4095 words larger if concurrent jobs are not active.

1.2.3.4 Write-Enable During Loading

When loading BASIC55 and DOSBASIC, the user was required to write-enable the disk to allow the program to write a necessary scratch file. Under BASICPLUS this is no longer necessary. The disk may be left write-protected.

1.3 The Notation Used in This Guide

The syntax of BASICPLUS is displayed using a few fundamental conventions:

* Words in all capital letters stand for themselves. Words in lower case letters are the names of other constructs.

* Corner brackets "<" and ">" are placed around words (in lower case) which name a class of items which can occur in place of the corner-bracketed name. For example, the manual specifies the item <digit> in a place where the user can enter any digit. Thus,

  <letter> <digit>

  means "any letter followed by any digit".

* Square brackets "[" and "]" are used to enclose optional items or groups of items. For example, if a digit may optionally be included, the notation would be [<digit>]. As another example, if the word "TO" may optionally be included, the notation is [TO].

* Ellipsis (that is, three consecutive periods "...") indicates "more of the same". For example:

```
INPUT <variable> [ , <variable> ]...
```

means "the word 'INPUT', followed by a variable name, followed optionally by one or more variable names separated from their predecessors by commas".

* The vertical bar "|" indicates an either-or choice. Thus,

```
<letter> [ <letter> | <digit> ]...
```

means "a letter, followed optionally by a series of one or more letters or digits".

* A group of definitions listed one above another also indicates an either-or choice.

CHAPTER 2. USING BASICPLUS

2.1 Installing BASICPLUS

BASICPLUS for 5500-type processors is distributed on both cassettes and diskettes. The 1500 version is available only on diskette. If BASICPLUS is received on a diskette, the program is ready to execute simply by inserting the diskette into an available drive. If BASICPLUS is received on a cassette, however, the following procedure must be used to catalogue the program onto a Disk Operating System disk:

1. Load the DOS system.

2. Place the cassette in the front deck.

3. Type: MIN ;A

When the MIN utility terminates, BASICPLUS is fully installed and ready for use. If you wish to rename BASICPLUS, type:

```
NAME BASICPLS/CMD,<name>
```

2.2 Loading BASICPLUS

To activate BASICPLUS, enter:

```
BASICPLS
```

2.3 Printer Configuration

BASICPLUS configures the on-line printer having the lowest address (servo, local, or remote). If no printer is found to be on-line, then no printer will be enabled.

To configure a specific printer (or to configure a servo printer for micro-plotting), type the command with additional options:

```
BASICPLS;<p> [ <line width> ]
```

where <p> is L, R, S, M, or N, and <line width> is the width of the printer line.

"L"	configures a local printer, default width: 132
"R"	configures a remote printer, default width: 79
"S"	configures a servo printer, default width: 132
"M"	configures a servo printer for micro-plotting, default width: 132
"N"	specifies that no printer shall be configured.

For example, if

```
BASICPLS;S100
```

was entered, a servo printer would be configured with a line width of 100 characters.

If a specific printer is requested, but is not on line, then a message to this effect will be displayed and no printer will be configured.

If no printer is configured, then the PRINT #4 command becomes illegal. Unused printer support area is automatically added to the user's work area.

No printing is allowed in columns beyond the line-width configured.

A remote printer is assumed to be 30 cps and 80 columns wide through the RS-232 connection of a 9400. When using a local or servo printer, make sure it is <u>on</u> and <u>on-line</u> during load.

Note: The 1500 version of BASICPLUS supports only serial printers connected to the processor printer channel. The program will not configure a printer unless specified by the "L" option (see above). The printer is assumed to be 132 columns wide. The printer does not need to be on-line during initializaton, but if PRINT statements are executed to a non-existent printer, BASICPLUS may hang.

2.4 BASICPLUS With Overlays

In order to allow more user space for larger programs, the user can load a special version of BASICPLUS which has overlays. The first overlay consists of the line scanner and the error messages. The second overlay consists of the execution routines.

While a program is being loaded from disk or typed in from the keyboard, the part of BASICPLUS which executes the program is not needed. While a program is being RUN, the part of BASICPLUS which scans lines is not needed. By having these two parts of BASICPLUS as overlays, the user acquires the extra space saved by not having both parts in memory at the same time.

The only disadvantage to using BASICPLUS with overlays is that loading overlays takes time. This version of BASICPLUS will load and execute programs with exactly the same speed as BASICPLUS without overlays, but there will be a noticeable pause when changing from scanning to executing or vice-versa, due to overlay loading.

To select the overlay version of BASICPLUS, type "O" after the semicolon on the command line in DOS. The "O" option may either precede or follow the printer configuration option, if one is required. Normal ways of loading the overlay version of BASICPLUS are:

```
BASICPLS;O
BASICPLS;OL
BASICPLS;NO
BASICPLS;OS115
BASICPLS;R1250
```

The message "OVERLAY VERSION" is displayed on the screen when that version of BASICPLUS is loaded.

CHAPTER 3. AN INTRODUCTION TO DATAPOINT BASICPLUS

Although this User's Guide is designed primarily as a reference tool, this chapter is an introductory tutorial to BASICPLUS. Ideally, the novice should be seated at a processor, keying text in and observing the response of BASICPLUS.

Type the command "BASICPLS" and press the ENTER key. This causes the Disk Operating System to load BASICPLUS. There will be a noticeable pause as BASICPLUS is being loaded and working areas are initialized.When the program is loaded, it displays a message identifying the version of BASICPLUS that you have loaded. Any messages to remind you of release changes are also displayed here. The next line tells which printer, if any, has been enabled. The next to the last line is a reminder that you are starting out with a clear workspace--that is, a workspace in the computer that contains no variable or programs.

The message "READY" indicates that BASIC is ready to accept a new command. Then the cursor, a little block of light that flashes on and off, appears on the screen. The presence of the cursor is your cue that BASIC is waiting for you to type something. When it says "READY" and flashes the cursor, it means that it wants you to type some command to the BASIC system.

Try typing

```
217*.06
```

and finish by pressing the ENTER key. BASIC interprets this as a command to calculate 217 multiplied by 0.06 and print the result. The answer can be interpreted as the interest on $217 at 6% for one year. This is an example of the desk calculator mode of BASIC. Any numeric expression typed in as a BASIC command will print the evaluated answer.

If you followed the above instructions, BASIC will have printed "READY" again and is flashing the cursor. To demonstrate the natural logarithm function, try typing

```
LOG (10)
```

which will print out the value of the natural logarithm of ten.

BASIC has many operations and functions, including:

+	Add
-	Subtract
*	Multiply
/	Divide
^	Raise to a power
SIN	Sine
COS	Cosine
TAN	Tangent
ATN	Arctangent
INT	Largest integer
EXP	Exponential
LOG	Natural logarithm
SQR	Square root
RND	Random number
ABS	Absolute value
SGN	Sign
MIN	Select the minimum of two expression values
MAX	Select the maximum of two expression values

Numeric values can be "remembered" by assigning those values to numeric variables. A numeric variable is a named entity whose value can be a number. If, for instance, you type

```
LET PI=3.1415927
```

you assign the value 3.1415927 to the variable named PI. The variable name PI can then appear any place that numbers could appear. Try typing

```
2*PI
```

to print out the value of two pi. A variable named PI need not have 3.1415927 as its value. Type

```
LET PI=-7
```

and the variable named PI has taken on the value -7. To prove it, type

```
PI
```

and BASICPLUS should print out -7.

Instructions to BASIC can also be stored as programs. Type

```
        10 P*(1+R/N)^N
```

and you have stored an instruction to calculate compounded interest. P is the name of a variable representing the principal, R is the name of a variable representing the yearly interest rate, and N is the name of a variable representing the number of times in a year the interest is compounded. The number 10 and the space at the beginning of the line tell BASIC that this is statement number ten. The actual number in this case is not important, but when there are many statements, the statement numbers specify the order in which they will be executed. Note that BASIC did not say "READY". This happens because BASIC is expecting another line of the program and it would be a nuisance to have the "READY" messages interspersed. Never fear, because commands are still legal.

You can now set up the values of the variables to be used.

```
        LET P=187
        LET R=.0575
        LET N=4
```

for a $187 principal at 5 3/4% interest compounded quarterly. You could now type

```
        P*(1+R/N)^N
```

and get the answer; but because you have stored that program you need only type

```
        RUN
```

and the processor will execute all the stored statements in numerical order. Since ten is the only stored statement, BASIC will evaluate the expression "P*(1*R/N)^N" and print the result.

To see what the result would be for 6% interest, merely change R by typing

```
        LET R=.06
```

and get the new result out by typing

```
        RUN
```

You might want to try changing the principal or the compounding frequency as well.

At this point you should feel in command of the desk calculator portion of BASIC. You have also written and stored a one-line program.

Prepare to start a whole new program by typing

```
SCRATCH
```

This erases the current program and variables and leaves you with a clear workspace. Into this clear workspace you will place a program that computes the time necessary for an object that falls off a desk to hit the floor. **Remember**, since these statements are numbered, they are only stored for later execution.

```
10 REM COMPUTE TIME NECESSARY FOR AN OBJECT FALLING
20 REM OFF A DESK TO HIT THE FLOOR
```

These lines, 10 and 20, are REMark statements that are used to document the program. A statement cannot be longer than one line, so the comment had to be divided into the two lines 10 and 20.

```
30 PRINT "THIS PROGRAM CALCULTES THE ELAPSED TIME"
```

Line 30 is a direction to print, when the program is RUN, the phrase between the quote marks. Unfortunately, "CALCULATES" was misspelled. This can be corrected by re-typing the line, then continuing:

```
30 PRINT "THIS PROGRAM CALCULATES THE ELAPSED TIME"
40 PRINT "REQUIRED FOR A HEAVY OBJECT TO FALL"
50 PRINT "FROM A DESK TO THE FLOOR."
```

Note that three separate PRINT statements were necessary to type out the entire message. If you make a mistake, re-type the line correctly. You can delete an entire line by typing only the line number. Continuing:

```
60 PRINT
```

will print only a blank line when it is run.

```
70 PRINT "WHAT IS HEIGHT OF THE DESK (FEET) ?";
```

will print out the material between the quotes (including the parentheses and question mark) when it is run. After it has done so, it will leave the cursor positioned at the end of the question. This is due to the semicolon at the end; without the semicolon, the cursor would appear on the line below the question.

```
        80 INPUT HEIGHT
```

This statement, when RUN, will start flashing the cursor and wait for you to type in a value for the variable named HEIGHT. The value you type in will become the value of the variable.

```
        90 LET TIME = SQR ( 2*HEIGHT / 32.2 )
```

This LET statement will compute a value for the variable named TIME. It will arrive at that value by multiplying the height in feet by 2, dividing by 32.2 (the gravitational constant in ft/sec squared) and then taking the square root of the entire quantity enclosed by the parentheses. Now there are two different variables: HEIGHT and TIME.

```
        100 PRINT TIME;" SECONDS"
```

Line 100 will print out the value of the variable TIME and then the literal " SECONDS" right next to it. The <u>value</u> of the variable named TIME is printed instead of the letters T,I,M, and E because there are no quote marks around it. The literal " SECONDS", on the other hand, is printed as such (blank, S, E, C, O, N, D, S) because it does have quote marks around it. The semicolon in the middle says to print the two items next to each other.

```
        110 END
```

The END statement identifies the end of the program.

You can get a listing of everything stored by typing

```
        LIST
```

Because that line does not begin with a line number it is executed immediately, producing a listing of the stored statements. Note that the program is listed in order of increasing line numbers even if you had to go back and correct some statements.

The listing of the program should be:

```
        10 REM COMPUTE TIME NECESSARY FOR AN OBJECT FALLING
        20 REM OFF A DESK TO HIT THE FLOOR
        30 PRINT "THIS PROGRAM CALCULATES THE ELAPSED TIME"
        40 PRINT "REQUIRED FOR A HEAVY OBJECT TO FALL"
        50 PRINT "FROM A DESK TO THE FLOOR."
        60 PRINT
        70 PRINT "WHAT IS HEIGHT OF THE DESK (FEET) ?";
        80 INPUT HEIGHT
        90 LET TIME=SQR(2*HEIGHT/32.2)
        100 PRINT TIME;" SECONDS"
        110 END
```

This program is now ready to go. To start it, type

```
        RUN
```

If everything is in order, it will print out the three lines of introduction, a blank line, and then ask the question of how high the desk is. As a good test, try

```
        16.1
```

as the response because that is the height at which it should take a full second. After BASIC prints the answer, it will show you that the execution ended at the END statement and that it is READY. All of this is normal. The values of variables are still available for inspection. Type

```
        HEIGHT
```

and BASIC will reply with 16.1 showing that 16.1 is the value of the variable named HEIGHT.

This program can be used to find out the fraction of a second needed for a penny to fall off your desk. Measure (or guess) the height of your desk in feet. RUN the program and enter the height. The answer shows how long it would take for the penny to fall from your desk to the floor.

The program can be made more convenient to use with a few changes in the input and output formats. For example, it would be much more convenient to specify the desk height in inches. You can do this with additional statements:

```
        80 INPUT HEIGHTININCHES
        85 LET HEIGHT = HEIGHTININCHES / 12
```

Note that you can put in spaces between parts of the statement, such as before and after the replacement sign (=) and the division symbol (/). These spaces are removed when BASIC reduces the statement to the most compact form possible for storage in the processor. You cannot, however, put spaces in a variable name like HEIGHT IN INCHES. Retype the query statement as:

```
        70 PRINT "WHAT IS HEIGHT OF THE DESK (INCHES) ?";
```

Now, RUN the program with this new change.

Because (for most desks) the time taken is a small fraction of a second, you might prefer to have it expressed in milliseconds (thousandths of a second).

```
        100 PRINT INT(TIME*1000);" MILLISECONDS"
```

This statement will (1) convert the TIME to milliseconds by multiplying by 1000; (2) use the INT function to eliminate the fractional part left after multiplication, leaving an integer, hence the name INT; and (3) print out the literal " MILLISECONDS" after the number of milliseconds. Note that the PRINT statement can print the result of a calculation.

If you now use

```
        LIST
```

to list this version of the program, you will see that the re-typed lines have been replaced. Try typing

```
        RUN
```

to see how this version works.

If there were several desks for which you needed this calculation, it might be handy to have a table of the results for common desk heights. One way to do this would be to RUN the program over and over again and write down the results. You can program BASIC to make the program work over and over again. After the last statement of the algorithm, you can place

```
        105 GO TO 70
```

and the program will go back to line 70 after printing out each answer. Therefore you can just type in desk heights and write down answers without typing RUN again and again. Try this out by typing:

```
        RUN
```

When you want to stop the program, the KEYBOARD key on the far right of the keyboard will do the job. Hold it down until the message "INTERRUPTED" is displayed. Pressing the KEYBOARD key will always enable you to interrupt a BASIC program. (See chapter 4 for futher information on the use of this key.)

The GOTO technique just demonstrated will allow you to print out as many values for the time as the number of heights you type in. Shouldn't a computer be able to make an entire table? Certainly. Type in these three lines:

```
        70 FOR HEIGHTININCHES = 30 TO 40
        80
        105 NEXT HEIGHTININCHES
```

Typing 80 alone deletes line 80 which used to ask for the input. The other two lines enclose an area of the BASIC program which will be repeated for values of HEIGHTININCHES from 30 to 40. Statement 105 indicates that it is time for BASIC to repeat the section with the next value for HEIGHTININCHES. Run this by typing

```
        RUN
```

There is only one problem remaining. We don't know which answer goes with which value of the height. This can be fixed with

```
        100 PRINT HEIGHTININCHES,INT(TIME*1000)
```

which will print each height and its time together on one line when executed. The comma in between the two values indicates that they should be placed in two columns (rather that one right after the other, as a semicolon would do). You can label the columns by adding:

```
        65 PRINT "HEIGHT","TIME"
        67 PRINT "INCHES","MSECS"
```

which will put headings at the top of the columns. Try this version by typing:

```
RUN
```

Would you like your answers on paper? Just change the PRINT statements to refer to the printer, device number 4.

```
65 PRINT#4,"HEIGHT","TIME"
67 PRINT#4,"INCHES","MSECS"
100 PRINT#4,HEIGHTININCHES,INT(TIME*1000)
RUN
```

You can also list a copy of the program on the printer by typing LIST#4. Note that the printer must be ON (and ON-LINE if a Local Printer) during initial loading of BASIC since this is when BASIC recognizes that there is a printer available. If you are using a 1500 processor, you must have specified the "L" option on the command line.

Welcome to BASIC programming, former novice! Of course, there's more to be learned. There are many other statements and operators avaiable in BASICPLUS. The remainder of this User's Guide discusses them all.

CHAPTER 4. MODES

BASIC may be waiting for a command, executing the instructions in a program, or requesting data for a program which it is executing. "Mode" refers to the state which BASIC is in during these activities.

4.1 Command Mode

BASIC enters the command mode when it has nothing further to do in order to accommodate the last command. BASIC is awaiting a command from the user (such as RUN, LIST, SCRATCH, or a desk calculator command) directing it to perform another task. This mode is indicated by the word "READY" and a flashing cursor.

BASIC is also in command mode after accepting a numbered statement to be stored. In this instance, BASIC does not prompt with "READY".

4.2 Input Mode

The input mode occurs during the execution of an INPUT statement in a RUNning program, when values for variables are requested from the operator. The cursor is flashing. Before an INPUT statement is executed, one or more PRINTstatements should be executed, specifying precisely what information is to be entered.

4.3 Running Mode

While BASIC is executing statements other than INPUT, the cursor is not visible and any keystrokes except KEYBOARD and DISPLAY are ignored. This is known as the running mode. If necessary, the KEYBOARD key may be used to interrupt the execution of the stored program (see the next section).

4.4 Changing Modes and Stopping the Display

Pressing the KEYBOARD key at any time will cause BASIC to re-enter command mode regardless of its current mode. Therefore, pressing KEYBOARD can be used to terminate input or to regain control from a runaway program. Hold down the key until BASIC responds with "INTERRUPTED". If BASIC was not in command mode already, the last statement executed in the current program is displayed.

The program can be continued by use of the GO TO command (See chapter 9).

Holding down the DISPLAY key at any time will suspend the program and prevent the screen from "rolling up", thereby "freezing" the display for examination. When the DISPLAY key is released, the program resumes and output continues.

CHAPTER 5. CONSTANTS AND VARIABLES

The most useful values in BASIC programs are those associated with variables. The value associated with a variable can be changed during the execution of a program. Some values in a BASIC program remain unchanged in all applications of the program. Such values are termed "constants".

5.1 Constants

Constants are values that do not change. There are two varieties: (1) Numeric constants, whose values are numbers and (2) String constants, whose values are "strings" of characters.

5.1.1 Numeric Constants

Numeric constants may be entered as integers or as real (decimal) numbers. Real numbers may also be specified in "scientific notation".

5.1.1.1 Integer Constants

Numbers may be entered in the form of signed or unsigned integers; however, all integers are converted by BASIC to floating point.

```
Form:  [ <sign> ] <digit> [ <digit> ]...
```

5.1.1.2 Real Constants

Real (decimal) numbers may also be specified. There is no restriction on placement of the decimal point in the real numbers.

```
Form:  [ <sign> ] <digit> [ <digit>... ] .
       [ <sign> ] . <digit> [ <digit>... ]
       [ <sign> ] <digit> [ <digit>... ] . digit [ <digit>... ]
```

In scientific notation, the decimal number (usually between 1 and 10) is followed by 10 to some power. The letter E replaces the "x 10" and the power of 10 follows the E.

Form: <decimal number> E [<sign>] <digit> [<digit>...]

5.1.1.3 Values of Numeric Constants

Any number may have as many digits as can fit on a line as long as the magnitude of the number is not greater than 1E38. When BASIC scans a number, however, it stores only the twelve most significant digits. In this way, the six digits that are displayed as output are correct. (See section 5.3 on implementation limits to values).

Examples of numeric constants:

1	has the value +1.0
+1	has the value +1.0
1.0	has the value +1.0
-1	has the value -1.0
-1.01	has the value -1.01
2345	has the value 2,345. Note: Commas are never used in numbers in BASIC
12E2	has the value 1,200 The E is read "times 10 to the"
5E+2	has the value 500
1.2E6	has the value 1,200,000
1E-2	has the value 0.01
-2.3E-4	has the value -0.00023

5.1.2 String Constants

String constants are quoted strings of characters: any sequence of characters (except the quote itself) delimited by quotes.

Form: " [<character>...] "

Note that the string of <u>no</u> characters is also allowed: "".

Because a statement cannot be continued on another line, the string length is limited only by the number of characters in the line.

Spaces within the quotes are treated the same as any other character in the line. For example, the string " Hi" is different from the string "Hi".

Examples of string constants:

"R"	has as its value, the letter R
"RS"	has as its value, the letters R and S
"1"	has as its value, the symbol known as "one"
"YES"	is the string constant for the 3 symbols that make up the word "YES"
"WHAT IS THE NAME OF THE 2ND BASEMAN?"	is a rather long string constant

String constants are totally defined by the characters they contain and the positions these characters occupy. That is, the constant "YES" can be analyzed by a BASIC program to determine the fact that it is 3 characters in length, that the first character is a "Y", that the second is an "E", and that the last is an "S".

5.2 Variables

Variables are associated with either numeric or string values. The value associated with a given variable can change as a program is executed. Because of their changeability, variables are referred to by names. BASIC variable names begin with a letter. Other consecutive letters and digits can follow, but embedded spaces are not allowed.

```
Form:          <letter> [ <letter> | <digit> ]...
```

String variable names are distinctive: they begin with a letter, continue with letters and digits, and end with a dollar sign (see chapter 12).

Variable names cannot begin with REM because those letters are reserved for inserting REMarks in programs. Any variable which starts with the letters "FN" and is followed by a single alphabetic character (See appendix E) is reserved for naming a user-defined function and is not allowable as a possible variable name.

All names reserved by BASIC for operators, functions, and commands are prohibited from being used as variable names. These reserved names are:

&	EF	IF	OPEN	STEP
ABS	EL	INPUT	OR	STOP
AND	END	INSERT	PRINT	SUB
APP	EOF	INT	RANDOMIZE	TAB
ATN	ERASE	INV	RD	TAN
AUTO	EXP	IOPEN	READ	THEN
BEEP	FALSE	LEN	REM	TO
BY	FF	LET	REN	TRN
CAT	FNn	LF	RESTORE	TRUE
CLICK	FOR	LIST	RETURN	UPDATE
CON	FREE	LOG	RND	USING
COS	GET	MAT	RU	VAL
CR	GO	MAX	RUN	VARS
DATA	GOSUB	MIN	SAVE	VP
DEF	GOTO	NEXT	SCRATCH	ZER
DELETE	HD	NEXTKEY	SGN	
DET	HP	NOKEY	SIN	
DIM	HU	NOT	SIZ	
DOS	IDN	ON	SQR	

Reserved names new to version 2 of BASICPLUS are:

INSTR	KEY	NOFILE	TOPEN	ROLLOUT

(See Appendix B, "RESERVED NAMES").

Examples of legal names for variables:

```
A
B4
BQ
CLASS
FOREVER73
YEARTODATEEARNINGS
YTDEARNINGS
CODEZEBRA9
```

The following are illegal variable names in BASIC:

7A	does not begin with a letter
NEG-	cannot have a dash
LET	cannot have a variable named the same as a BASIC operator or command

```
        REM8    cannot begin with REM (special rule)
```

5.3 Advanced Naming Techniques

BASIC variable names may be of any length but are implicitly limited because a statement cannot be broken over lines. The following symbols are also considered alphabetic for use in variable names: $ and _ (dollar and underline). See appendix E for a list of valid alphabetic characters. If a variable name ends with a $, it is considered to be a string variable (see chapter 12).

Lower case letters may be used in variable names, but the names so formed are distinct from those with different casing (i.e., A1 and a1 are the names of two different variables). In the same way that many BASIC implementations restrict variable names to <letter> [<digit>], many also treat upper- and lower-case letters as identical in variable names.

Since all BASIC reserved words are composed solely of upper-case letters, variable names which include lower-case letters or digits or both will never confict with them.

5.4 Implementation Limits on Values

All numeric values are stored internally in Datapoint BASIC as floating-point numbers with one byte (8 bits) of characteristic and five bytes (40 bits) of mantissa. All floating point numbers are kept normalized at all times and the sign is therefore taken to be the complement of the most significant bit of the mantissa. Zero is the only unnormalized number permitted. As a result of this representational scheme, very small numbers which could ordinarily be expressed as an unnormalized number with the smallest exponent cannot be represented in BASIC.

Since all numbers (including integers) are converted by BASIC to floating point, it is the user's responsibility to convert the results of calculations to integers if integer values are required. To guarantee integer results, the program should round values:

```
        240 NUM=INT(X+.5)
        250 PRINT NUM,"CHILDREN IN AVERAGE CLASSROOM"
```

The largest number representable is approximately 1E+38. The smallest positive number representable is approximately 1E-38.

Precision is ideally 40 *(log base 10 of 2) or 12.04 digits. however, values are rounded to 6 digits on output to conform to previous versions of Datapoint BASIC. Note that subtracting similar numbers will lead quickly to a loss of precision. Zero fill is used in normalization.

Overflows during numeric expression evaluation are reported with the message "OVERFLOW". The characteristic has exceeded the maximum size. Underflow is not announced, and the result is set to zero.

CHAPTER 6. STATEMENTS

BASIC statements can be stored for later execution or executed immediately. If a BASIC statement is preceded by a line number (between 1 and 38399) followed by a space, that statement will be stored under that line number. If the statement begins without a line number, BASIC will attempt to execute the statement immediately. This latter mode is useful for commands and desk calculator operations.

6.1 Spacing

Spaces are important in BASIC programs; they are used to delimit the elements of statements. For example:

```
        LETCOUNTER=7
```

sets the value of LETCOUNTER to 7 (the assignment verb, LET, being implied) but

```
        LET COUNTER=7
```

sets the value of COUNTER to 7.

Where one space can appear, many may. Spaces are also legal between parts of statements as in

```
        LET COUNTER =       7 + 9 * ( 2 * 8 )
```

Spaces may not appear within a single variable name or within a BASIC reserved word. The following is ILLEGAL:

```
        LE T COUNT ER = 0
```

because spaces occur in the middle of LET and COUNTER.

If a BASIC program is created using EDIT, it is stored "as is," blanks and all. A program is more readable when its FOR-NEXT loops are indented, for example. When such a program is loaded by BASICPLUS (see the GET Command in chapter 20), spaces around operators and spaces in excess of one are removed (except within quoted strings) to increase storage capacity within the interpreter.

When a program held by BASIC is copied out to disk (see the SAVE Command in chapter 20), it is copied "as is," in blank-stripped form.

6.2 Remark Statements

Form: REM [{any sequence of characters }]

Examples:
```
REM THIS PROGRAM CALCULATES THE INNER PRODUCT
REM THE NEXT SECTION CALCULATES THE INTEREST
REMBRANT WAS A GREAT PAINTER
REM PRINT "HI THERE"
```

The REM statement allows comments within a BASIC program. Comments make the program more readable. They do, however, take up space that could otherwise be used for active statements.

The REM statement is completely ignored when it is executed. Even the PRINT in the last example will have no effect. Putting valid BASIC characters inside a REM comment is as effectively ignored as any other information.

Only the first three letters need be REM. NO space is necessary after the REM. This is an exception to the general rule that spaces must be present between statement parts.

6.3 Multiple-Statement Lines

A multiple-statement line is a line which contains more than one instruction, with the instructions separated by semicolons. Multiple statements are useful at the beginning of a stored program to give initial values to the variables:

```
10 LET A=0; B=0; PI=3.1416; I=1
```

In this case, the multiple statement is equivalent to:

```
10 LET A=0
20 LET B=0
30 LET PI=3.1416
40 LET I=1
```

Another particularly useful feature of multiple statements is in the IF statement:

```
40 IF A=B THEN C=C+1;D=D+1;PRINT "FOUND A MATCH"
```

This statement is not equivalent to the three separate statements: 'IF A=B THEN C=C+1', 'D=D+1' and 'PRINT "FOUND A MATCH"'. If these three statements were on three successive lines, D would be incremented and the message printed regardless of whether A and B were equal. Using the multiple statement, C and D will be incremented and the message printed only if A=B. If A does not equal B, the rest of the line is ignored.

If any of the following statements (or commands) occurs in a multiple-line statement, it must be the last element of that line:

GOTO, GOSUB, SCRATCH, DOS, RUN, APP, GET, STOP, END, RETURN, REM

The following are correctly formatted:

```
LET OneMore=OneMore+1;A=1;B=2;C=3;D=4
10 LET OneMore = OneMore + 1; A=B=C=0
20 DIM String$[3];LET String$[1]=9;String$[3]=12
Pi=355/113;A=A+1;IF A<12 THEN B=B-1
```

CHAPTER 7. NUMERIC EXPRESSIONS AND THE ASSIGNMENT (LET) STATEMENT

Numeric expressions in BASIC are combinations of numeric operators, numeric functions, and numeric variables and constants. Numeric operators and functions can operate upon:

Numeric constants
Numeric variables
Parenthesized numeric expressions

and to be discussed later:

Fully subscripted vectors and arrays
Fully subscripted strings or string vectors
Substrings or substrings of string vector elements

7.1 Numeric Expressions

The numeric operators are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation (Raising to a power)
MIN--arithmetic	Select the minimum of two values
MAX--arithmetic	Select the maximum of two values

The numeric functions (including the transcendentals) are:

INT	Select the largest integer below argument
LOG	Natural logarithm
EXP	Exponential
SIN	Sine
COS	Cosine
TAN	Tangent
ATN	Arctangent
SQR	Square root
RND	Random number
ABS	Absolute value
SGN	Sign

7.2 Precedence of Operations

The precedence of numeric operations, from greatest to least, is:

```
( <numeric expression> )
numeric functions
^
* and /
MAX and MIN
+ and -
```

7.2.1 Examples

2+3*4	Note that the multiplication is carried out before the addition. If in doubt, use parentheses to force order of evaluation. Result is 14.
(2+3)*4	Parentheses forced 2+3 to be evaluated first. Result is 20.
(2+5/6	Illegal because a right parenthesis is missing.
INT(.8)	Zero. INT finds the largest whole number that is less than or equal to the value of its argument. INT(1) is 1. INT(1.253) is 1. INT(-1.9) is -2.
SQR(4)	Two.
SIN(.1)	9.98334E-2 (very close to 0.1). Argument of SIN and COS is in radians. Likewise, ATN returns a result in radians. To convert degrees to radians, multiply by 3.14159/180.
4^2	Four squared. Four is raised to the second power.
SQR(4+12)	Four, the square root of 16.
SQR(4)+12	Fourteen. The square root is taken first and then twelve is added.
RND	Random decimal fraction between 0 and 1. A dummy argument is optional, but not required. It is ignored if present.

SGN(-.3)	Negative one. SGN(X)=0 if X=0; SGN(X)=1 if X is positive and SGN(X)=-1 if X is negative.
ABS(-.5)	Positive one-half.
24 MIN 6	Six. The minimum of the two arguments is taken.
24 MAX 6	twenty-four. The maximum of the two arguments is taken.
2 ^ 3 ^ 2	Sixty-four. The two is cubed, then the result is squared.
2^(3^2)	Five hundred twelve. The three is squared, then the two is raised to the ninth power.

7.3 Functions

Form: <function> (<numeric expression>)

7.3.1 Numeric functions

INT	Gives the largest integer that is less than or equal to the value of the numeric expression. Unlike previous Datapoint BASICs, BASICPLUS places no limit on the value of the argument.
SQR	Gives the non-negative square root of the numeric expression.
RND	Gives the next pseudo-random number in a sequence of pseudo-random numbers uniformly distributed in the range 0 <= RND < 1. The numeric expression is a dummy argument, which is optional, and is ignored if present. The random number sequence is reset to the first number in the sequence each time a RUN command is given.
ABS	Gives the absolute value of the numeric expression.
SGN	Gives the sign (-1, 0, +1) of the numeric expression.

7.3.2 Transcendental functions

Note that the argument of the trigonometric transcendental functions is given _in radians_.

LOG	Gives the natural logarithm of the numeric expression value.
EXP	Raises e (e=2.71828182846) to the power of the numeric expression value.
SIN	Gives the sine of the numeric expression value.
COS	Gives the cosine of the numeric expression value.
TAN	Gives the tangent of the numeric expression value.
ATN	Gives the arctangent of the numeric expression value.

7.4 The Assignment (LET) Statement

Form:
```
[LET] <variable> = <numeric expression>
```

Examples:
```
LET A=19
LET TOTAL=PIECE1 + PIECE2
MANE = LION / FUR
```

An assignment or replacement statement is used to assign a value to a variable. The value may be a constant or it may be computed as a numeric expression involving any of the numeric operations. Parentheses may, of course, be used to form a complicated expression.

There is only one case where the word "LET" must appear: if a multiple statement contains an assignment following a PRINT statement, the word LET must follow the semicolon. Otherwise, the word LET is optional. For example:

```
PRINT A;B;LET C=1
```

would print the values of A and B, after which C would be assigned a value of one.

```
PRINT A;B;C=1
```

would print the values of A and B, followed by either TRUE or FALSE depending on whether or not C equals one (see boolean expressions, section 9.3.2).

7.5 The RANDOMIZE Statement

Form:	RANDOMIZE
Example:	RANDOMIZE

As noted before, each time the RUN command is used, the random number generator is reset to the first number in the sequence. This is useful for debugging programs which use the RND function, so that the same random number sequence may be used over and over again. However, once a program has been debugged it is often desirable to use a different random number sequence (or, at least, start at a different spot in the same random number sequence). This is the purpose of the RANDOMIZE statement. This command resets the random number seed to a random value. If this statement is used at the beginning of each program, then the random number sequence will not be the same each time the RUN command is given.

The RANDOMIZE statement consists solely of the word "RANDOMIZE" with no parameters.

CHAPTER 8. USER-DEFINED FUNCTIONS

```
Form:        DEF <function name> = <numeric expresssion>

where <function name> is defined as:

             FN <letter>

Examples:    DEF FNA= SIN(X)/X
             DEF FNZ=FNA+FNB
```

The user-defined functions are very similar to subroutines, except that in many cases, they are easier to use. They are also similar to the system-defined functions discussed earlier, but they cannot have true parameters. Once a function is defined using a DEF statement, it may be used anywhere a numeric expression can be used. When a numeric expression contains a function reference, the value of the function is computed (according to the definition) and is "substituted" for the reference in the expression. The function may call other functions, up to a limited depth. For example:

```
10 DEF FNA=X^2
20 FOR I=1 TO 10
30 LET X=I
40 PRINT I,FNA
50 NEXT I
```

will display:

```
1             1
2             4
3             9
.             .
.             .
10            100
```

The above program is equivalent to the following one:

```
10 FOR I=1 TO 10
20 GOSUB 60
30 PRINT I,X
40 NEXT I
50 END
60 X=I^2
70 RETURN
```

If a function is referenced without ever having been defined, the message "UNDEFINED FUNCTION" is displayed.

CHAPTER 9. CONTROL STATEMENTS

Control statements allow for the interruption of the normal sequence of execution of statements by causing execution to continue at a specified line, rather than the one with the next higher line number.

9.1 The GOTO Statement

```
Form:          GO TO <line number>
               GOTO <line number>
               GO <line number>
               TO <line number>

Examples:      GO TO 120
               GOTO 90
               GO 65
               TO ABC
               GOTO (100*I)
```

The GO TO statement transfers control of execution unconditionally. Ordinarily, the BASIC program is executed in line number order. Encountering this statement changes that order; execution continues at the specified line.

The <line number> can be specified as a number which represents a line. If, upon execution, no such line exists, the message "NO SUCH LINE" will be printed and execution will halt. <line number> can also be a variable which has as its value the number of the line to be executed next. Expressions are also legal if they are enclosed in parentheses as in the last example. If a numeric expression is used, its value is truncated to an integer before examination.

9.1.1 GOTO Without A Line-Number

If <line number> is omitted, the first line of the program is executed. Thus, the command "GO" is useful to start programs if the initialization performed by the "RUN" command is not desired. This is often the case if one is working extensively in command mode and enters running mode only as a aid to the direct execution.

9.2 The GOSUB and RETURN Statements

```
Form:          GOSUB [<line number>]
               GO SUB [<line number>]
               SUB [<line number>]
               RETURN

Examples:      GOSUB 19
               GOSUB (10*I)
               GO SUB 11
               SUB
               RETURN
```

The GOSUB and RETURN statements can be used to access subroutines in BASIC. The GOSUB behaves exactly like a GO TO except that the statement number of the GOSUB is recorded. When a RETURN statement is executed, execution continues at the line following the one containing the GOSUB.

9.2.1 GOSUB Nesting

GOSUBs can be nested so that subroutines call subroutines to a limited depth. The execution of the GOSUB and RETURN statements can be described in terms of a stack of line numbers. Prior to the execution of the first GOSUB by the program, this stack is empty. Each time a GOSUB is executed, the line number of the GOSUB is placed on top of the stack and execution of the program is continued at the line specified in the GOSUB. Each time a RETURN is executed, the line number on top of the stack is removed from the stack and execution continues at the line following the one with that line number. It is not necessary that equal numbers of GOSUB and RETURN statements be executed before termination of the program.

9.2.2 GOSUB Without a Line-Number

GOSUB without an argument will GOSUB to the first statement of the program.

9.2.3 Debugging With GOSUB

GOSUB can be used in direct execution mode to debug a subroutine. Set up applicable variables. GOSUB <line number> in direct execution mode will cause the subroutine to be executed. At the RETURN statement, control is returned to the user. Variables and/or output can be examined for correct operation.

RETURN executed as a direct command will remove memory of the last GOSUB from BASIC and otherwise act as a no-operation.

9.3 The IF Statement

Form:

```
IF <condition> THEN <line number> [;ELSE <line number>]
IF <condition> THEN <BASIC statement> [;ELSE <BASIC statement>]
```

<condition> is defined as:

```
              <relation>
              (<relation>)
              NOT <relation>
              <relation> AND <relation>
              <relation> OR <relation>
```

<relation> is defined as:

```
              <expression> <relational operator> <expression>
              <expression>
```

<relational operator> is defined as:

<	(less than)
<=	(less than or equal to)
=	(equal to)
>=	(greater than or equal to)
>	(greater than)
<>	(not equal to)
#	(not equal to)

Examples:

```
        IF COST>10000 THEN 50
        IF COST>10000 THEN PRINT "Cost too high"
        IF A THEN PRINT "A IS NON-ZERO"
        IF NOT A THEN PRINT "A EQUALS ZERO"
        IF A-7=B+9 THEN A=B;D=E;ELSE A=A-1;BEEP
        IF A THEN C=1;D=2;ELSE 50
        IF A=B THEN (23+C)
        IF A<B AND A<C THEN 19
        IF NOT A<=B THEN PRINT "A greater than B"
        IF N1=1 AND N2=2 OR N3=(1+2) THEN LET FOUR=4
        IF (A=B OR C=((SIN 5)/D)) THEN PRINT TAB(9);
        IF A<1 THEN IF B>=2 THEN IF C=3 THEN END
```

The IF statement allows the programmer to make a decision based on the values of boolean expressions composed of variables and constants. Comparisons can be made using the operators <, <=, =, #, <>, >=, and >. These conditions can be compounded using the operators NOT, AND, and OR. Parentheses may be used to specify the order of complicated conditions.

Decisions can also be based on a zero/non-zero value. A zero value is interpreted as a FALSE condition, whereas a non-zero value is interpreted as a TRUE condition.

If the condition is true, BASIC examines the part of the statement after the word THEN. If the word THEN is followed by what could be a line number, execution continues at the line specified by the line number. If BASIC determines that the construction is not a line number, the remainder of the statement up to an ELSE or the end of the line is executed and (if no control statements in the THEN clause direct otherwise) execution then continues at the line following the IF statement.

When the condition is false, BASIC searches the line for an ELSE clause. If an ELSE clause is found, BASIC inspects the item following the ELSE. If this is a line number, execution continues at the line specified by the line number. If the item following the ELSE is not a line number, then the remainder of the statement is executed and (if no control statements in the ELSE clause direct otherwise) execution continues at the line following the IF.

9.3.1 Testing for Exact Equality

Because of the limited precision of computer representation for numbers two expressions which, in absolute precision, produce equal results may produce slightly unequal results in BASIC. If the results are compared as:

```
IF A = B
```

BASICPLUS will determine that they are <u>unequal</u>. Therefore, it is more advisable to compare for equality within a given tolerance, i.e.:

```
IF ABS(A-B) < .000005 THEN <statement>
```

9.3.2 Boolean Expressions

Boolean algebra is the algebra of logic. Boolean expressions have only two values: TRUE and FALSE. In BASICPLUS, TRUE is equivalent to 1 and FALSE is equivalent to 0. All of the comparison operators used in the IF statement

```
=, <, >, <=, >=, <>, #, NOT, AND, OR
```

can be used in building boolean expressions. Boolean expressions evaluate to TRUE or FALSE (i.e., 1 or 0). An example or two can eliminate much explanation:

```
(7=43) has the value 0
(84 > 1) has the value 1
TRUE has the value 1
FALSE has the value 0
```

Boolean expressions can be used in place of any numeric expression. For example:

```
LET C=(A=B)
```

would set C to 0 if A<>B and set C to 1 if A=B.

Values can also be assigned using other operators:

```
C=C+A*(A<B)
```

would increment C by A if A<B, but would leave C untouched if A>=B.

9.3.2.1 The KEY Condition

The KEY condition is set true when a key has been depressed on the keyboard and has not been read in. This flag may be used during tight loops to determine if the user wishes to enter data. This flag has a boolean value of TRUE or FALSE (1 or 0) depending on whether or not a key has been pressed. Note that the key is not removed from the keyboard buffer and will be echoed when an INPUT statement is executed. Examples of the use of KEY are:

```
     IF KEY THEN INPUT A1,A2,A3
     KEYREADY = KEY
     IF NOT KEY THEN 10
```

9.3.2.2 Boolean Expressions in Command Mode

Conditions can be tested for truth in Command mode. Type the boolean expression; BASICPLUS will reply with "TRUE" or "FALSE" as appropriate. The words "TRUE" and "FALSE" are also valid conditions in themselves. Therefore, typing "NOT TRUE" prints out "FALSE". IF statements can often be debugged using this capability.

<u>Restriction</u>: the "=" relation is not available for this use unless the relational condition is enclosed in parentheses because it conflicts with the use of "=" in direct assignment statements. (This is because, in Command mode, the command A=B is an assignment statement, not a condition.) Use "(A=B)" or an equivalent relational condition like "NOT A<>B" instead.

9.4 The ON GOTO Statement

```
Form:          ON <num exp> GOTO <line number list> [ ; <stmt> ]

<line number list> is defined as:

               <line number> [ , <line number> ]...

Examples:      ON A GOTO 10,20,30
               ON (C+1) GOTO 1,2,3,4;PRINT "***ERROR***"
               ON 3 GOTO 300,200,100,400,500
```

In the ON GOTO statement (sometimes referred to as the multi-branch GOTO), <num exp> indicates which of the specified statements will be executed. For example statement 20, above, transfers control to statement 100 because 100 is the third line number in its list.

The value of the <numeric expression> is truncated to an integer, which is then used to select a line number from the list following the GOTO. The line numbers in the list are indexed from left to right, starting with one. Execution of the program continues at the line with the selected line number.

If the expression evaluates to a number greater than the number of line numbers specified, or less than one, the GOTO is ignored unless a semicolon is present following the list. In this case, the statement following the semicolon will be executed. For example, "***ERROR***" would be printed in the second example if (C+1) evaluated to a number greater than four or less than one.

9.5 The ON GOSUB Statement

```
Form:           ON <num exp> GOSUB <line number list> [ ; <stmt> ]

<line number list> is defined as:

                <line number> [ , <line number> ]...

Examples:       ON (Z+1) GOSUB 5,3,47,26
                ON Z9 GOSUB 10,20;PRINT "* ILLEGAL Z9 *"
```

The ON GOSUB statement (sometimes referred to as the multi-branch GOSUB) is identical to the ON GOTO statement except that, instead of a GOTO being executed to the specified statement, a GOSUB is executed. As in the ON GOTO, all expressions are truncated.

If the numeric expression evaluates to an integer less than one, or greater than the number of statements specified, the optional statement following the list, separated by a semicolon will be executed. If this statement is not present, the GOSUB will be ignored.

A statement to trap indexing errors in the ON GOSUB (i.e., an executable statement following the list of expressions) should almost always be used, because there is no other way to detect such errors. For example, if an indexing error did occur, then execution would fall through to the next line. However, if an indexing error did not occur, then execution would transfer to the specified subroutine and when the RETURN statement was executed control would then return to the statement following the ON GOSUB. Thus if the error did occur, it could not be detected. The execution would continue at the line following the ON GOSUB line in either case.

CHAPTER 10. THE FOR AND NEXT STATEMENTS

```
Form:    FOR <simple num var> = <start> TO <final> [STEP <incr>]
         FOR <simple num var> = <start> TO <final> [BY <incr>]

         NEXT <variable>

<start>, <final>, and <incr> are <numeric expression>s.

Example: FOR ZZ=1 TO 19
         . . . . . .
         NEXT ZZ
```

The FOR and NEXT statements allow repeated execution of the statements between the FOR and the NEXT. The group of statements from a FOR to its NEXT is sometimes referred to as a FOR-NEXT loop.

The <simple num var> is a simple numeric variable (i.e., a scalar numeric variable, not a subscripted array name) which will be changed each time the statements contained in the loop are executed. It will commence at <start>. It will grow by 1 every time through the loop unless the STEP clause is used to specify another value for <incr>. As soon as the value of the <variable> exceeds the value of <final>, the loop will end and execution will continue with the statement following the NEXT statement. The index <variable> retains its (incremented) value.

10.1 Forward Loops and Backward Loops

Loops can run either "forwards" or "backwards". The default value for <incr> is +1 and the loop runs forwards with the <variable> getting larger every time. If the value of <incr> is negative, <variable> gets smaller each time.

On forward-running loops, iteration continues until the value of <variable> exceeds that of <final>. When the loop is exited, <variable> contains this value. If the loop runs backwards, the loop iterates until <variable> becomes smaller than <final>.

10.2 Modifying Loop-Controlling Variables

Both <final> and <incr> can be modified by the statements within the loop during the execution of the loop. For example:

```
FOR K=1 TO 50 STEP K
```

will execute the loop 6 times, as K takes on the values 1, 2, 4, 8, 16, and 32. At the exit of the loop, K will have the value 64. The value of <final> may be similarly manipulated within the loop. The value of <variable> may also be changed within the loop with predictable results. Note that it is possible to have a loop that runs forward at some times and backward at other times if the sign and value of the STEP and <final> expressions are changed.

10.3 Nested FOR-NEXT Loops

FOR-NEXT loops may be nested. The innermost loops are completed first. A transfer of control out of the range of FOR is legal.

Executing a corresponding NEXT statement (i.e., a NEXT whose <variable> is the same as that of the most recent FOR) will cause the loop to be repeated regardless of the lexical context of the NEXT. If the loop is exhausted, execution will continue at the NEXT which most immediately follows the FOR in lexical order regardless of the position of the NEXT which caused the loop to be iterated.

10.4 Active and Inactive FOR-NEXT Loops

The number of active FOR loops is limited. A FOR loop is active if a NEXT statement is legal for that loop. This is true for every loop entered unless it was exited by exhaustion or another loop utilizing the same loop variable was entered. Therefore, problems may be encountered if many FOR loops are exited with GO TOs. These problems can be avoided by coding new loops with the same iteration variable as loops exited with GO TOs. This precaution is not needed except in extraordinary programs.

10.5 Examples of FOR-NEXT Loops

The loop below will print 1, 1.5, 2, 2.5, 3, 3.5 ...... to 10.

```
10 FOR LOOPVARIABLE=1 TO 10 STEP .5
20 PRINT LOOPVARIABLE
30 NEXT LOOPVARIABLE
```

The loop below will print nothing because the loop is vacuous. See also the next example.

```
10 FOR B=10 TO 1
20 PRINT B
30 NEXT B
```

The loop below will print out the values from 10 down to 1.

```
10 FOR B=10 TO 1 STEP -1
20 PRINT B
30 NEXT B
```

The loop below will only print the value 10.

```
10 FOR B=10 TO 10
20 PRINT B
30 NEXT B
```

CHAPTER 11. OUTPUT STATEMENTS: PRINT, BEEP, AND CLICK

These three statements are used to communicate to the keyboard operator.

11.1 The PRINT Statement

```
Form:           PRINT [ <print list> ] [ <separator> ]

<print list> is defined as:

                [ [ <print item> ] <separator> ]... [ <print item> ]

<print item> is defined as:

                <expression> ¦ TAB ( <numeric expression> )

and <separator> is defined as:

                <comma> ¦ <semicolon>

Examples:       PRINT "THE TABLE OF PRIME IMPLICANTS FOLLOWS:"
                PRINT SIN(ANGLE1)
                PRINT COUNTER;" TIMES"
                PRINT "ENTER THE PRINCIPAL AMOUNT: ";
                PRINT 4+5,(9^SQR(2)),14
                PRINT "SOURCE","RESULT",X,Y,Z
```

This form of the PRINT statement is used to print information on the CRT display. (Another form of the PRINT statement can output to disk or printer.) The statement specifies a list of variables, constants, and expressions whose values are to be printed. Both numeric and string values are permitted. The word PRINT by itself produces a blank line. All 80 columns of each screen line are accessable to the PRINT statement.

The separators between the values to be printed determine the format of the output:

```
                , Comma means "next zone"
                ; Semicolon means "no spacing"
```

The print line is divided into 5 zones, each 16 spaces wide. Separating items with commas tells BASIC to move into the next available zone to output the next value. Even if the next print item is a TAB, the "move to next zone" is acted upon before the TAB is considered. Semicolon means no spacing; the next value will be printed adjacent to the current value.

Each PRINT statement causes one line to be printed unless:

* the values will not all fit on one line, in which case extra lines will be used; or

* a comma or semicolon terminates the statement, in which case the next PRINT statement will continue where the current one stopped.

In order to use the trailing-separator form of the PRINT statement in a multiple-statement line, the terminating separator (, or ;) must be followed by a semicolon before the next operator. Example:

```
    PRINT; PRINT; PRINT
```

will print three blank lines, whereas:

```
    PRINT "A=";;LET A=3*3;PRINT A
```

will print:

```
    A= 9
```

If the screen is full and an attempt is made to PRINT more lines, the display "rolls up"--the top line disappears, the remaining lines move up, and the new line appears at the bottom. Pressing the DISPLAY key "freezes" the display, stopping any more PRINTing until the key is released.

Other forms of the PRINT statement are described in chapter 17, "File Input and Output".

11.1.1 The TAB Function

Form: TAB (<numeric expression>)

Examples:

```
PRINT TAB (4),"Name";TAB (26),"Address"
PRINT TAB (66);
PRINT 5+6;TAB(LOG(A)),EXP(1.02+A)
```

The TAB function can be used only within a PRINT statement. It causes the next value to be printed beginning in a specified column number. The columns are numbered from 1 to 80.

If TAB is preceded by a comma separator, the current print position is <u>the beginning of the next print zone</u>. Therefore, if the PRINT statement was:

```
PRINT A,TAB (8),B
```

the value of B would begin at column 17 rather than at column 8.

The TAB reference must be followed by a separator if more instructions follow on the same line. The output need not be done in the same PRINT statement; see the second example. The last example shows TAB being used to squeeze output together while still leaving more spaces than a plain semicolon would have left.

If the argument to TAB is less than one or greater than 80, the message "I/O ERROR" is displayed. <u>No action is taken if the TAB position specified is to the left of the current print position, but greater than 0</u>. Note that a comma follows the tab function if more expressions follow in the same PRINT statement. This <u>does not</u> cause zonzed output to be used, and is required by the syntax of the PRINT statement.

11.1.2 Advanced Display Techniques

The contents of the Datapoint processor display can be manipulated and randomly accessed with BASICPLUS. For this purpose, special formatting functions are used within the PRINT statement. These functions are, like the TAB, valid only within the PRINT statement.

HP (X)	- Position to the specified horizontal position
VP (X)	- Position to the specified vertical line
EF	- Erase from the cursor to the end of the screen

EL	- Erase from the cursor to the end of the line
RU	- Roll the screen up one line
RD	- Roll the screen down one line
HU	- Home up (position to upper left corner)
HD	- Home down (position to lower left corner)

The argument to HP may range from 0 (left edge) through 79 (right edge). The argument to VP may range from 0 (top of twelve line screen) to 11 (bottom of twelve line screen), or from -12 (top of twenty-four line screen) to 11 (bottom of twenty-four line screen).

All of the above instructions must be followed by a comma if more instructions follow on the same line. HP and VP may be followed by a semicolon if on the end of the line. All other functions may be followed by either a semicolon or nothing at the end of a line depending on the type of formatted output desired.

For example, the following will clear the screen and position a message in the middle:

```
10 PRINT HU,EF,HP (30),VP (5);
20 PRINT "D a t a p o i n t   B a s i c p l u s"
30 GO TO 30
```

The method of cursor positioning used under DOSBASIC is still valid in BASICPLUS. In this method, special code values in strings are used to direct special actions.

```
08 - A new horizontal position follows
       Note: For the 1500 version, this value is 9.
11 - A new vertical position follows
17 - Erase to the end of the screen
18 - Erase to the end of the line
19 - Roll up one line
20 - Roll down one line
03 - End of string
13 - End of string with return and line feed
```

For example, the following will clear the screen and position a message in the middle:

```
10 DIM CLR$(5)
20 CLR$(1)=11;CLR$(2)=0;CLR$(3)=8;CLR$(4)=0
30 CLR$(5)=17
40 PRINT CLR$
50 CLR$(2)=5;CLR$(4)=30
60 PRINT CLR$;"D a t a p o i n t   B a s i c"
```

Note: It is the user's responsibility to be sure that formatted output uses valid control characters and cursor positioning for the type processor the program is being executed on. For example, roll downs are not legal on a terminal running under PS, and while vertical position -1 may be legal for a twenty-four line screen, it is not legal for a twelve line screen.

11.2 The BEEP and CLICK Statements

```
Form:      BEEP
           CLICK
```

The BEEP and CLICK statements cause the processor to beep or click, respectively. These are used for signalling the keyboard operator that a certain point in the program has been reached, that an error has occurred, or that an input response is expected. The click is preferred in situations where the beep would be annoying.

CHAPTER 12. THE INPUT STATEMENT

Form: INPUT [<variable> ,]... <variable>

Examples:

Program specifies:	Operator Keys In:
INPUT A	3.14159
INPUT A,B,CCC	2 -3,4.526
INPUT X, Y$, Z(4)	15 HELLO 3
INPUT TRIALNUMBER	345

The INPUT statement is used to request information from the operator. The cursor on the CRT display begins flashing and the keyboard is activated for input. Values are taken from the line typed in by the operator and assigned sequentially to the variables in the INPUT statement.

Each INPUT statement accepts one line of input from the operator. If the operator types in fewer values than there are variables in the INPUT list, the remainder of the variables are given the value zero. If too many numbers are typed in, the excess is discarded.

Numeric input is free-format; any preceding blanks are ignored. Any character that cannot appear within a number (such as blank or comma) terminates the number and the next number begins in the position after the terminating character, if the terminating character was a comma.

The calculation of subscripts is done "on the fly," so that if the program:

```
10 DIM A(20)
20 LET I = 3
30 INPUT I,A(I)
```

is executed, the element of array A that is inputted depends not on the previous value of I (namely 3), but the value of I just inputted before the element A(I). This feature simplifies the entry of elements into relatively sparse arrays.

12.1 Strings in INPUT

String data is entered without quotes. If quotes are entered, they are included as part of the string value. Furthermore, INPUT does not ignore preceding blanks which are input to strings; these are assigned as characters in the string variable.

Use caution when mixing numeric and string data. For instance: The number of characters that are considered as input to the string variable is determined by the length of the string given in its DIM statement.

```
10 DIM S$(10)
20 INPUT A, S$, C
```

If the operator keys in:

```
5,HELLO,7
```

then the values received are: A=5, S$="HELLO,7 ", and C=0. In order to input a value to C, the operator should have keyed in:

```
5,HELLO     7
```

The best way to avoid confusion during string input is to have separate INPUT statements for each string variable. For example:

```
10 DIM S$(10)
20 INPUT A,C
30 INPUT S$
```

In this way, no mistake can be made as to what is numeric and what is string input.

Other forms of INPUT statements are described in chapter 17, "File Input and Output".

CHAPTER 13. THE READ AND DATA STATEMENTS, AND RELATED MATTERS

```
Form:       READ <variable> [ , <variable> ]...
            DATA <value> [ , <value> ]...
            RESTORE
            EOF

Examples:   READ A,B,C
            DATA 10.32,-4,19E5
            RESTORE
            IF EOF THEN PRINT "NO MORE DATA"
```

The READ statement behaves much like the INPUT statement except that data is retrieved from DATA statements rather than from the keyboard.

The values in all the DATA statements in a program, taken in line number order, constitute a single sequence of data for the READ statements of the program. Each READ obtains one value from the sequence for each variable in its list. The values in the DATA statements match the type of the variable used in reading. In other words, if the variable in the READ list is a numeric variable, the next value in the DATA sequence must represent a numeric value; if a string variable (see Chapter 16), then a string value. The individual DATA statements are not directly associated with the individual READ statements, therefore there need not be any correspondence between the number of variables named in a particular READ statement and the number of values in a particular DATA statement. Expressions are not allowed in DATA statements.

The RESTORE statement causes the data to be re-read from the beginning.

If the user READs beyond then end of the DATA sequence, the error message "NO MORE DATA" will be displayed. The user may test for this situation. EOF is a condition which can be tested in the IF statement. It returns TRUE if the DATA sequence has been exhausted, and FALSE otherwise. A READ executed which cause EOF to come TRUE will receive zeros for numeric items and the null string for string items.

Example:

```
    DATA 1,2,3,4.5,040,45E3
    READ X,Y,Z             X gets the value 1, Y=2, Z=3
    READ Q,R               Q gets the value 4.5, R gets the
                           value of 40
    RESTORE
    READ Z                 Now Z gets the value 1
    READ X,Y,Z,A,B
    IF EOF THEN 20         No action is taken
    READ A                 A gets the value 0
    IF EOF THEN 20         Transfer is made to line 20
```

13.1 Strings in DATA Statements

String values may be specified in the DATA statement in quoted form.

```
    DATA "JOHN","DOE","JANE","TEXAS",3.4,"234-89-3678"
```

If the quotes are absent, BASICPLUS perceives a sequence of letters as a variable name.

CHAPTER 14. THE STOP AND END STATEMENTS

```
Form:      STOP [ " <comment> " ]
           END [ " <comment> " ]

Examples:  STOP
           STOP "ABC is out of range"
           END
           END "ALL DONE"
```

The STOP and END statements are used to signal the end of stored program execution. Control reverts to the keyboard operator. If a comment in quote marks follows the STOP or END, it will be printed with the STOP or END when encountered. This provides a means of assuring the operator of proper completion. It can also be used as a fatal error message indicating why the program stopped early. Good programming form dictates that the last statement of every program must be an END statment (unless the program is to be used in a DOS chain; See Program Chaining, Chapter 21).

The END statement is also used to signal that transactions with a file are completed and to free the file number to be used with another file (see File Input and Output, Chapter 17).

Executing the STOP statement is equivalent to pressing the keyboard key, in that BASICPLUS stops running the program and enters COMMAND mode. Execution can be resumed with the GOTO <linenumber> command. END performs the functions of STOP, but it also forces input or output from or to disk files to be completed and the files to be closed.

CHAPTER 15. ARRAYS, THEIR DECLARATION AND MANIPULATION

In addition to ordinary (scalar) variables, each of which represents a single value, BASIC provides for lists and tables (array variables). A list, or vector, can be thought of as a column of numbers. A table, or matrix, can be regarded as consisting of entries having both rows and columns.

15.1 The DIM Statement

```
Form:      DIM <variable> <bounds> [ , <variable> <bounds> ]...
```

<bounds> is defined as:

```
           ( <bound1> [ , <bound2> ] )
```

and both <bound1> and <bound2> are <numeric expression>s.

```
Examples:  DIM A(19)
           DIM B(24,4)
           DIM C(5),D(6,7),E(99)
           DIM FFF(4*C),GGG(EXP(Q1))
```

The DIM statement is used to indicate:

1. that the variable named is an array
2. the number of dimensions the array has
3. the maximum extent of each dimension of the array.

The <bounds> specify the number of dimensions and the extent of each. The first example above allocates storage for a singly-dimensioned array named A having 19 elements. When two bounds are specified, they designate the number of rows and columns in the array. The second example dimensions an array of 24 rows by 4 columns.

In a one-dimensional array, numbers are stored and referenced as indexed (or subscripted) members of a list. For example, to designate the third element of the array A, use A(3). The parentheses are used to enclose the subscript.

In a two-dimensional array, numbers are stored and referenced via two indices, the row and the column respectively. The subscripted variable D(2,5) references the number in the second row, fifth column of array D.

BASICPLUS converts "(" and ")" to "[" and "]", to aid in distinguishing the parentheses of subscripts from the parentheses of numeric expressions and of function arguments. The user may subscript with parentheses or brackets, but BASICPLUS always performs this conversion. When BASICPLUS GETs a program from a file, it performs this conversion. If the program is SAVEd to a file, the subscripts are delimited by brackets regardless of how they were entered.

The value of the <bound>s can be any numeric expression which can be evaluated at the time the DIM is performed. This includes expressions involving the results of previous calculations. If the value of the numeric expression is not an integer, it is truncated to an integer.

A scalar variable which has previously contained a value cannot be redefined as an array and vice-versa. BASICPLUS reports an error when the programmer uses the same name for what must be different and independent variables (since they are of different types). The message "VARIABLE ALREADY DEFINED" will be display if the user tries to DIM a already define variable, and the message "ARGUMENT NOT NUMBER" will be displayed if the user tries to use a DIMed variable in a scalar application.

Some BASIC implementations do not allow dimensioning of arrays by expressions, requiring instead constant dimensions. BASICPLUS allows dimensioning of arrays by expressions, provided that the array has not already been dimensioned. Executing the same DIM statement more than once in a program or executing another DIM statement naming the same array will cause the error message "VARIABLE ALREADY DIMENSIONED" to be printed. If this occurs, the program must be modified so that the array is dimensioned only once. <u>An array which has been DIMed cannot be DIMed again until a new program is used or a RUN command is given</u>. If an array must be re-DIMed during the execution of a program, the RUN command with a non-existant line number for an argument will re-initialize array storage and allow new DIM statements.

The message "NO ROOM" occurs when there is no space in the processor memory to allocate space for an array or a program. Occasionally, enough space can be recovered by a SAVE and GET to allow execution. Alternatively, removing other arrays from array space by executing the RUN command to a non-existent line will give enough space to allocate the current array.

The DIM statement is legal as a direct execution statement. To procede with execution of the stored program, use GOTO <linenumber> rather than RUN, since RUN deletes DIM-allocated storage. Make sure, however, the <linenumber> is past the DIM statement.

In BASICPLUS, unlike some other BASIC implementations, the DIM statement must be executed before the array is actually created.

15.1.1 Use of Arrays

A subscripted variable can appear any place where a variable can appear except as the index for a FOR-NEXT loop. The subscript may involve any numeric expression evaluable at the time the subscripted variable is encountered. Here are some examples:

```
10 DIM A(20),B(4,5)
20 LET A(2)=10
30 DIM C(A(2)-1)
40 A(A(2))=A(2)
50 IF A(2) <> A(SQR(100)) THEN PRINT "Oops!"
60 B(1,1)=5
70 FOR I=C(1) TO B(1,1)
80 C(1+I)=B(I,I)+I
90 NEXT I
100 PRINT A(2),B(1,1),C(6)
```

15.2 Matrix Operations - The MAT Statement

Many useful functions for array processing can be done in BASICPLUS using built-in matrix operations to set all elements of an array to zero or one, to set up an identity matrix, to copy arrays, and so on.

15.2.1 The Matrix Assignment Statement

```
Form:     MAT <matrix1> = <matrix2>

Examples: MAT A=B
          MAT MATRIX1=MATRIX2
```

The matrix assignment statement copies each element of

<matrix2> to the corresponding element of <matrix1>. The two matrices must have the same dimensions (previously defined in a DIM statement) or an error will occur. If the two matrices have the same dimensions, then <matrix1> will become an exact copy of <matrix2>.

15.2.2 The Matrix Arithmetic Statements

Form: MAT <matrix1> = <matrix2> <operator> <matrix3>

<operator> is defined as:

```
+ for matrix addition
- for matrix subtraction
* for matrix multiplication
```

Examples:

```
MAT A=B+C
MAT Z9=A1*B1
MAT X=Y-Z
```

The matrix arithmetic statements are used to perform arithmetic operations on matrices: addition, subtraction, and multiplication (+,-,*).

Under addition and subtraction, all matrices must have the same dimensions.

The rules of matrix multiplication are:

If <matrix2> is a P x Q matrix,
then <matrix3> is a Q x N matrix,
and the resulting <matrix1> is a P x N matrix.

Thus, for MAT C = A * B,
if matrix A is 5 x 3, then
matrix B must be a 3 x n matrix.
If, for example, it is 3 x 7,
then the resulting matrix will be 5 x 7.

If these conditions are not met, an error results. The same matrix must not appear on both sides of the equal sign for multiplication or an error will result.

15.2.3 Scalar Multiplication of a Matrix

```
Form:       MAT <matrix1> = ( <scalar> ) * <matrix2>

Examples:   MAT A=(5)*B
            MAT Z=(A+B)*Z
```

This operation facilitates multiplication of a matrix by a scalar value (possibly a numeric expression value). Both <matrix1> and <matrix2> must have the same dimensions or an error will result. Each element of <matrix2> is set equal to the corresponding element of <matrix1>, multiplied by the scalar. The same matrix may appear on both sides of the equals sign.

15.2.4 The MAT INPUT Statement

```
Form:       MAT INPUT [ # <num expr> , ] <matrix> [ , <matrix> ]...

Examples:   MAT INPUT A
            MAT INPUT#2,Z
```

The MAT INPUT statement is used to obtain values for one or more entire matrices from the keyboard or a disk file (see chapter 17, File Input and Output). The absence of a file number expression or the file number being equal to 0 indicates that the MAT INPUT is to obtain its matrix values from the keyboard.

Each matrix is entered in row-major order, the second subscript varying most rapidly [e.g., A(1,1), A(1,2), A(2,1), A(2,2), B(1,1), B(1,2), etc.] Note that a MAT INPUT statement reads only <u>entire</u> matrices. If a matrix is dimensioned to have 50 elements, a MAT INPUT of that matrix demands all 50 elements.

If not enough data is entered from the keyboard to fill the matrix, the message "NOT ENOUGH DATA, ENTER MORE:" will be displayed and a new line of data will be requested.

If not enough data is present in a file to fill the matrix, a "NO MORE DATA" statement will be displayed.

If the format of the data is incorrect, the message "BAD INPUT, RETYPE FROM ITEM:" will be displayed. This message does not mean that the entire line was rejected, it simply means that the user must recognize his mistake and re-enter data from that point. For example:

```
DIM A(5)
MAT INPUT A
1,2,3,4,5
MAT INPUT A
1,2
NOT ENOUGH DATA, ENTER MORE:
3,4
NOT ENOUGH DATA, ENTER MORE:
5
MAT INPUT A
5,6,7M8,9
BAD INPUT, RETYPE FROM ITEM:
8,9
MAT PRINT A;
5 6 7 8 9
```

15.2.5 The MAT READ Statement

```
Form:       MAT READ [ # <num expr> , ] <matrix> [ , <matrix>]...

Examples:   MAT READ A
            MAT READ B,D,C9
```

The MAT READ statement is used to obtain values for one or more entire matrices from DATA statements or a disk file (see chapter 17, File Input and Output).

The absence of the file number expression or a file number being equal to 0 indicates that values are to be obtained from the DATA sequence (see chapter 13).

As with the MAT INPUT statement, matrices are entered in row-major order and only entire matrices are read. If not enough data is present in DATA statements to fill the matrix, the matrix will be filled with as much data as is available, and the message "NO MORE DATA" will be displayed. The remainder of the matrix will be left <u>untouched</u>. As in the READ statement, if an attempt is made to read invalid data into the matrix, an error will result.

Example:

```
10 DATA 1,2,3,4,5,6,7,8,9,10
DIM A(5)
MAT READ A
MAT PRINT A;
1 2 3 4 5
MAT READ A
MAT PRINT A;
6 7 8 9 10
RESTORE
READ B,C,D
MAT READ A
MAT PRINT A;
4 5 6 7 8
MAT READ A
NO MORE DATA
MAT PRINT A;
9 10 6 7 8
```

15.2.6 The MAT PRINT Statement

Form: MAT PRINT [# <num expr> ,] <mat print list>

where <mat print list> is:

             <matrix> [<sep> <matrix>]... [<sep>]

Examples:

```
MAT PRINT A
MAT PRINT A;
MAT PRINT A,B,C
MAT PRINT A;B,C
```

The MAT PRINT statement is used to output one or more matrices to the CRT display or to a disk file. The absence of the file number or the file number being equal to 0 indicates that the output is to be to the CRT display. The separators (comma or semicolon) control the formatting, as in the PRINT statement. The program:

```
10 DIM A(5,5)
20 MAT A=IDN
30 MAT PRINT A
```

is equivalent to the program:

```
10 DIM A(5,5)
20 MAT A=IDN
30 FOR I=1 TO 5
40 FOR J=1 TO 5
50 PRINT A(I,J),
60 NEXT J
70 PRINT
80 PRINT
90 NEXT I
```

If a semicolon follows the matrix name, the matrix will be printed in a compressed format. This would be comparable to a change in the second program of:

```
50 PRINT A(I,J);
```

More than one matrix may be printed with the MAT PRINT statement. They may be separated by either commas or semicolons, depending on the format desired.

15.2.7 The MAT ZER Function

Form: MAT <matrix> = ZER

Examples:
```
MAT A=ZER
MAT MATRIX1=ZER
```

The MAT ZER function is used to set all elements of a matrix to zero. This is often useful in logical operations, as FALSE is a logical zero. The matrix specified must already be DIMensioned.

15.2.8 The MAT CON Function

Form: MAT <matrix>=CON

Examples:
```
MAT Z9=CON
MAT MATRIX2=CON
```

The MAT CON function is similar to the MAT ZER function, except that all elements of the matrix are set to one. This is useful for setting all elements of a matrix to a logical TRUE. The same rules that apply for MAT ZER apply for MAT CON.

15.2.9 The MAT IDN Function

```
Form:       MAT <matrix>=IDN

Examples:   MAT A=IDN
            MAT MATRIX3=IDN
```

The MAT IDN function returns an identity matrix. An identity matrix is a square matrix with all diagonal elements equal to one and all off-diagonal elements equal to zero. Any matrix multiplied by an identity matrix equals itself. The same rules that apply to MAT ZER and MAT CON apply to this function, except that only a square matrix may be defined as an identity matrix.

Example of MAT IDN:

```
10 DIM A(4,4)
20 MAT A=IDN
30 MAT PRINT A;
```

Running this program produces:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

15.2.10 The MAT TRN Function

```
Form:       MAT <matrix1> = TRN ( <matrix2> )

Examples:   MAT A=TRN B
            MAT Z =TRN(Z)
            MAT X9=TRN(A)
```

The MAT TRN function returns the transpose of a matrix. The argument, <matrix1>, must have the same number of rows as <matrix2> has columns, and the same number of columns as <matrix2> has rows. For example, if <matrix1> is 4 x 8, then <matrix2> _must_ be 8 x 4.

A square matrix may be transposed into itself.

Example:

Original Matrix:

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

Transposed matrix:

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

15.2.11 The MAT INV Function

Form: MAT <matrix1> = INV (<matrix2>)

Examples:

```
MAT A=INV B
MAT X1=INV(X1)
MAT Z9=INV(A0)
```

The MAT INV function returns the inverse of a _square_ matrix. If the matrix is not square, or <matrix1> does not have the same dimensions as <matrix2>, an error will result. A matrix may be inverted into itself, as in the second example.

15.2.12 The DET Function

Form: <variable> = DET (<matrix>)

Examples:

```
A=DET(B)
Z(1)=DET(MATRIX1)
```

The DET function returns the determinant of a square matrix. A single value is returned as the determinant of the matrix.

Note: The determinant and inverse routines require a temporary workspace the size of the matrix plus the size of one column. The error message "INSUFFICIENT CALCULATION SPACE" will be displayed if not enough workspace can be found.

CHAPTER 16. STRINGS

Preceding chapters have discussed the use of BASIC with numeric data. This chapter describes the string-handling capabilities of Datapoint BASICPLUS. A string of characters can represent a name, a heading, a sentence, or any other kind of character data.

16.1 String Constants

A string that does not change is called a string constant. A string constant is written in BASIC by enclosing it in quotation marks.

```
          PRINT "THIS IS A CONSTANT STRING"
```

The PRINT statement above prints the constant string consisting of the 25 characters beginning with the "T" (of "THIS") and ending with the "G" (of "STRING").

16.2 String Variables

BASIC programs do not deal solely in numeric values; the ability to manipulate string values is integral to BASIC. The presence of string variables enables the user to write programs which accept, manipulate, and output character text. The names of numeric variables begin with a letter and continue with letters and digits. String variable names have the distinguishing mark of always ending with a dollar sign "$".

In BASIC, numeric values always occupy the same amount of storage (twelve digits, plus exponent). Since string values can range from no characters (the null string) to 251 characters, the user must specify how much space is to be allocated for each string variable. This is done using the DIM statement.

16.2.1 Simple String Variables

BASICPLUS treats simple string variables as arrays of characters. The DIM statement specifies the number of characters a string may hold. The value associated with a string variable cannot exceed its dimensioned maximum length. The program:

```
10 DIM MESSAGE$(80)
20 LET MESSAGE$="NOW IS THE TIME FOR STRINGS"
30 PRINT MESSAGE$
```

will print out the value of MESSAGE$, namely:

```
NOW IS THE TIME FOR STRINGS
```

Note that the quotes are not part of the string value.

16.2.2 String Lists

A one-dimensional array (a list) of strings is represented as a two-dimensional array of characters. When a string is given two dimensions in the DIM statement, the first dimension specifies the number of strings and the second specifies the maximum number of characters in each string. For example:

```
DIM A$(15), B$(11,20)
```

specifies that A$ is a string of up to 15 characters and B$ is a list of 11 strings, B$(1)...B$(11), of up to 20 characters each.

16.3 Substrings

Form: <string name> (<num exp> , <num exp> [, <num exp>])

Examples:
```
B$(4,10)
C$(1,4,7)
```

A substring is a portion of the string associated with a string variable. BASICPLUS provides a notation for designating substrings, making their use more convenient in both string expressions and string assignments.

The notation for the substring of a simple string is its name followed by a parenthesized pair of values designating the starting and ending points in the named string. The first example above shows a reference to the fourth through tenth characters of string B$.

The notation for a substring of a member of a string list is the name of the string list followed by a parenthesized list of three values designating first the member of the list, and then the starting and ending points in that member. The second example above designates the substring of the first member of string list C$, from its fourth through seventh characters.

If the beginning and ending points are equal, a one character substring is being specified. Here are some examples of the substring in use:

```
R$=F$(2,M)
AH$(S(1),S(2))="SUBSTRING"
PRINT C$(1,3,10);" IS THE VALUE."
```

BASICPLUS allows a shorthand way of specifying a single character substring. Instead of specifying equal beginning and ending points, a single value may be specified. Thus, if A$ is a simple string variable then A$(3) is equivalent to A$(3,3). Similarly, if L$ is a string list then L$(4,8) is equivalent to L$(4,8,8). Since simple strings cannot have the same names as string lists, no confusion arises from this shorthand.

The values for the starting and ending points must be greater than or equal to one, and less than or equal to the dimensioned length of the string, respectively. If the value for the starting point is less than one, or the value for the ending point is greater than the dimensioned length of the string, or the value for the starting point exceeds the value for the ending point, a subscripting error occurs.

16.3.1 The Equivalence of Characters and ASCII Values

Computers store strings in a coded form. Each character of a string is represented by a numeric code from a code table. In Datapoint computers, the ASCII (American Standard Code for Information Interchange) code is used. For example, the letter "A" is stored as 65 and the character " " (blank) is stored as 32. Appendix C gives the ASCII code values of the commonly used characters. BASICPLUS handles all the conversions between codes and characters.

BASICPLUS allows the user to manipulate the individual characters of a string as numeric values. A one-character substring can be assigned a numeric value, or participate in a numeric expression. Thus E$(1,1) = 65 would set the first character of string E$ to "A". As another example,

```
R$(1,1) = G$(2,2) + 5
```

would set the first character of R$ to the value of the character from the ASCII table which is five positions beyond that of the second character in string G$. If G$ had the value "THANKS", and R$ had the value "WASHING", executing the above statement would take the "H" of "THANKS", compute the character "M" from it, and give R$ the value "MASHING".

16.4 Assignment to Strings and Substrings

When a string is assigned a value having more characters than its dimensioned length, the excess characters are lost. When a string is assigned a value with fewer characters than its dimensioned length, a special character is assigned to the next position of the string. This special character is called ETX (End of TeXt) and has the ASCII code value of 003.

The ETX serves BASICPLUS as an end-of-string marker. The PRINT statement stops printing a string value when it encounters an ETX or a CR (Carriage Return = 13). The LEN function (below) measures string length up to an ETX or the end of the allocated space, which ever comes first. Under string concatenation (below), each string value being concatenated ends with the character before any ETX or at the end of the allocated space.

Because of this convention the assignment of values to substrings, such as in LET or INPUT statements, may produce results which are not at first obvious. For example, the program:

```
10 DIM A$(10)
20 LET A$="HELLO THERE"
30 PRINT A$
40 LET A$(1,3)="999999999"
50 PRINT A$
60 LET A$(1,3)="22"
70 PRINT A$
80 END
```

prints out:

```
HELLO THER
999LO THER
22
```

In line 20, the final "E" was lost. In line 40, only three of the "9"s were inserted. In line 60, an ETX was inserted because only two characters were replacing three. Thus statement 70 stopped printing after the "22". Actually, the "LO THER" remains in memory. If a single character (say, "*") were inserted at A$(3,3), a subsequent PRINT statement would again display the full ten characters.

The same string variable can appear on both sides of the assignment operator "=". For example, it is legal to concatenate a blank onto the front of a string or concatenate a string with itself using the LET statement (see String Concatenation below).

16.4.1 String Input and Output

Strings of no more than 80 characters can be entered or displayed via the CRT. Any string may be output to a printer. Strings of not more than 249 characters may be input from (or output to) disk files.

On output, a string is written up to its dimensioned length unless a CR character (Carriage Return = ASCII value 13) or an ETX character (End of TeXt = ASCII value 3) occurs in the string, in which case the output ends with the character preceding the CR or ETX.

The INPUT statement will accept as many characters as are specified by the string or substring argument. If fewer characters are entered than specified, an ETX is appended to the characters. If more characters are entered than specified, the excess characters are ignored. Chapter 12 (The INPUT Statement) further discusses the behavior of strings in the INPUT statement.

16.5 String Expressions

As numeric constants and variables may be combined with arithmetic operators to form numeric expressions, so may strings and substrings be combined with string operators to form string expressions.

The string operators and their effects are:

+ --string	Concatenates two strings together
USING	Edits one or more values into a string
MIN--string	Selects the lesser of two strings
MAX--string	Selects the greater of two strings
INSTR--string	Searches for one string within another

There are two functions which accept string expressions as arguments and return numeric values:

LEN (A$)	Returns the number of characters in A$
VAL (A$)	Returns the numeric value for which A$ contains the numeric representation.

A string constant or variable or some combination of strings, substrings, and string operators constitutes a string expression. The functions, since they return numeric values, participate only in numeric expressions even though they have string expressions as arguments.

16.5.1 String Comparison

The values of two (or more) string expressions may be compared in the IF statement or with the string MAX and string MIN operators. The two string values are compared, character by character.

If the Nth character of the first string is not identical to the Nth character of the second, the values of the codes of the two characters determine which is the greater.

If one string has fewer characters than the other, but the characters of the shorter match the characters of the longer up to the end of the shorter, then the longer one is the greater.

If the strings have the same number of characters and have the same characters in corresponding positions, then they are equal.

For example:

```
"CART" < "CART "              "CART" MIN "CART " = "CART"
"CAT" < "DOG"                 "CAT" MAX "DOG" = "DOG"
"CAT" < "DO"                  "CAT" MIN "DO" = "CAT"
```

16.5.2 String Concatenation

```
Form:       <string expr> + <string expr> [ + <string expr> ]...

Examples:   PRINT "HELLO"+"THERE."
            STRING$="YOU ENTERED "+NAM$+"IN 'NAME'."
```

The "+" operator, when used with entire strings in this manner, is not the arithmetic addition operator but rather the concatenation operator. Concatenation is the joining together of a series of strings in sequence. If A$ contains "CAT" and B$ contains "ALOG", then:

```
            A$ + B$
```

yields "CATALOG". No other arithmetic operators may be applied to entire strings.

16.5.3 String Editing (The USING Operator)

```
Form:       <format string> USING <exp1> [ , <exp2>]...
            PRINT USING <format string>, <exp1> [ , <exp2>]...
```

<format string> is defined as:

```
            <string expression>
```

```
Examples:   A$="THE COST WAS $#####.##." USING COST
            PRINT USING A$,VAR1,VAR2,VAR3
            B$=("@### $##,##@.## USING A,B)+A$
            OPEN#1,"FILE#/SRC" USING N
```

The USING operator enables the BASICPLUS programmer to edit a series of values to produce a string of characters. This feature is most useful for generating neatly formatted output. The USING operator may occur in any string expression. Another form of the USING, shown in the third example above, is provided for compatibility with other BASIC systems.

Editing is governed by the formatting fields in the format string. These fields consist of combinations of the following characters:

```
            # $ * @ ^ ~ , .
```

Other characters are printed exactly as they appear in the string. Each character in a format string reserves space for one character in the resulting string value.

16.5.3.1 Numeric Fields

16.5.3.1.1 Integer Field

The first of the two fundamental numeric formatting fields is the integer field, specified by a series of two or more number signs. Each number sign reserves space in the resulting character string for one digit of the integer. The value being edited is rounded to an integer and right-justified in the field. In other words,

```
"###" USING 46.58
```

produces:

```
 47
```

that is; a blank, a four, and a seven. The blank indicates a positive value. For a negative value, a minus sign (-) is used in place of the blank. The minus sign is placed directly to the left of the left-most digit. A plus sign is never actually printed. Thus, the string expression:

```
"##########" USING -7
```

yields "-7" preceded by 8 blanks.

16.5.3.1.2 Fixed Point Field

The other fundamental numeric formatting field is the decimal, or fixed point, field. As in the integer field, number signs specify the positions where decimal digits are to appear in the result string. The character "." indicates the position of the decimal point in the field; that is, the decimal point determines the justification. The value being edited is rounded, if necessary, to fit the field. For example, the string expression

```
"####.#####" USING 47.2
```

produces the character string

```
47.20000
```

Trailing zeros are generated as needed to the right of the decimal point, but leading zeros are replaced by blanks (that is, "blank-filled"). The sign is generated as in the integer field: in the character position preceding the left-most digit, a blank for a positive value or a minus sign for a negative value.

16.5.3.1.3 Zero-Filled Field

The commercial-at "@" is used to retain leading zeroes. If the "@" is used _in place of_ any number sign of an integer or fixed point field, for example:

```
PRINT USING "@#####.##", 75
```

then leading zeroes will be edited into the field positions which would otherwise be blank-filled:

```
000075.00
```

The leading position contains "0" if the value is positive, and "-" if the value is negative.

If the "@" is imbedded in the field, then the zero filling will begin at the place of the "@".
For example

```
PRINT USING "##@##.##", 7.5
```

will cause leading zero suppression to stop and zero filling to begin at the position of the "@".

```
007.50
```

16.5.3.1.4 Asterisk-Filled (Check Protect) Field

The asterisk "*" is often used in check protection. The asterisk-filled field is similar to the "@" field discussed above, but instead of generating leading zeros, it generates leading asterisks. For example, the result of:

```
"*########.##" USING 4765.99
```

is

```
          ******4765.99
```

The leading position contains "*" if the value is positive, and "-" if the value is negative.

This is used for formatting negotiable items, in order to prevent numbers from being inserted in front of the original amount ("4765.99" could be changed to "994765.99").

16.5.3.1.5 Comma Insertion Field

Formatting can also place commas in large numbers to aid readability. If commas are included in integer, decimal, zero-filled, or asterisk-filled fields, they will be edited into the formatted value. For example, the statement:

```
          PRINT USING "###,###,###",45636477
```

will cause " 45,636,477" to be displayed. If a digit does not precede a comma in the formatted field, then a blank (or zero or asterisk, as appropriate) is inserted instead of a comma.

16.5.3.1.6 Floating Dollar Sign Field

A numeric field <u>preceded</u> by a dollar sign "$" is said to have a "floating dollar sign"; that is, a dollar sign is generated immediately to the left of the left-most digit of a positive value or immediately to the left of the minus sign of a negative value. For example:

```
          PRINT USING "$######.##",4721.42
```

displays:

```
            $4721.42
```

with two leading blanks.

The dollar sign can be prefixed to the integer, fixed point, zero-filled, asterisk-filled, or comma-insertion fields. The statement:

```
          PRINT USING "$###,###,###.##",AMOUNT
```

with AMOUNT equal to 47521.7 will display:

```
$47,521.70
```

on the screen.

16.5.3.2 Scientific and Engineering Notation Fields

The "^^^^" appended to an integer or fixed point field specifies that the value shall be edited in scientific (that is, powers of ten) notation. For example:

```
PRINT USING "####.####^^^^",45621
```

displays

```
4.5621E+04
```

The format "~~~~" is almost identical to the "^^^^", except that it produces <u>engineering</u> notation. In engineering notation, the exponent is always a multiple of three (e.g. 0, 3, 6, 9, and so on). Note that <u>exactly</u> four (4) carets or tildes are used. These are replaced by "E+nn" or "E-nn". If the caret or tilde occurs in any other manner in a format string, it is treated as a literal, to be copied verbatim.

16.5.3.3 Character Field

Numeric values need not be the only type of data that is formatted. String values can also be formatted using a series of number signs. For example:

```
PRINT USING "### IS YOUR NAME","JOE"
```

displays:

```
JOE IS YOUR NAME
```

Unlike numeric values, string values are left-justified in their fields with blank-fill to the right.

16.5.3.4 Values Too Large for Their Formats

If a numeric value to be formatted is too large for the field specified, BASICPLUS indicates this problem with a question mark in the sign position. Here is an example:

```
          PRINT USING "###.##",475623.495
```

displays: "?47562". As much of the number as the format permits is generated in the field following the tell-tale question mark. The high-order digits are generated at the expense of the low-order digits.

If a string value to be formatted is too large for the field specified, it is truncated on the right, and no indication of the overflow is provided.

16.5.3.5 Composite Formats

A format string may consist of a sequence of fields. For example:

```
          "### ###.### #######" USING 1,2.4,6
```

yields

```
             1    2.400       6
```

The number of items in the USING list must exactly equal the number of fields in the format string. Any characters present in a format string which are not part of a field, are taken literally. Any characters except the number sign (#) or commercial at (@) (blanks, or even the formatting symbols ".", "$", "*", "^", and "~") may be placed anywhere in the format string and thus be edited into the resulting string value. The formatting symbols are treated as formatting symbols only if they appear within an integer or fixed point field (the "$", "*", and "."), or scientific notation or engineering notation field (the "^", "~", or "."). For example:

```
          PRINT USING "$ * . V = #####",12.45
```

will display:

```
          $ * . V =     12
```

and

```
     PRINT USING "I BOUGHT ### CARS at $####.## EACH.",37,4751.8
```

will print:

```
          I BOUGHT  37 CARS at $4751.80 EACH.
```

16.5.3.6 Additional Precision Displayable

All calculations are carried out to twelve decimal places of accuracy, however values are only displayed to six places in an ordinary PRINT statement. The numbers may be edited to display all 12 places by using formatted printing. For example:

```
          PRINT USING "#.############",2/3
```

will cause

```
          .666666666667
```

to be displayed.

16.5.4 The MIN and MAX Operators

```
Form:      <string expression> MIN <string expression>
           <string expression> MAX <string expression>

Examples:  A$[1]=X$ MIN "A"
           PRINT F$ MAX "DATUM"
           A$="UN"+(C$(2,5) MAX Z$)
```

String values may be compared with one another with the MIN and MAX operators, as mentioned above under String Comparison. The MIN and MAX operators are used to compare two strings and select the greater or lesser, respectively.

16.5.5 The INSTR Operator

```
Form:      <string expression> INSTR <string expression>

Examples:  IF A$ INSTR B$ THEN 10
           C="LOCATE" INSTR COMMAND$
           D=DATE$ INSTR "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
```

The INSTR operator searches the right-hand string expression for an occurane of the left-hand string expression. The result obtained is the character position of the right-hand (searched in) string. If the left-hand (search) string does not appear anywhere in the searched in string, a value of zero is returned.

16.5.5.1 Examples of String Search (INSTR)

```
     "*" INSTR "ABCDEFGHIJ*KLMNOP"
```

Would return a value of 11.

```
     "HELLO" INSTR "NOW IS THE TIME TO SAY HELLO TO SOMEONE."
```

Would return a value of 24.

```
     WHERE="NOTIN" INSTR "THIS STRING IS SEARCHED."
     PRINT WHERE
```

Would print the value "0".

16.5.6 The LEN Function

```
Form:      LEN ( <string expression> )

Examples:  PRINT A$[3,LEN(A$)]
           L=LEN (STRING1$ + STRING2$)
```

The LEN function returns a count of the number of characters in a string expression up to the first ETX (ASCII code 3) or the dimensioned length of the string, which ever comes first. When a string is first dimensioned, its length is zero; that is, the DIM puts an ETX in the first allocated position of the string. If fewer characters are assigned to a string than its DIM allocates, an ETX is placed after the last character assigned.

16.5.6.1 Examples of LEN in Use

```
10 DIM ANSWER$(3),NAME$(10)
20 PRINT "This program prints backwards."
30 PRINT "Type in the name: ";
40 INPUT NAME$
50 FOR I=LEN (NAME$) TO 1 STEP -1
60 PRINT NAME$(I);
70 NEXT I
80 PRINT " Do you want to try another? ";
90 INPUT ANSWER$
100 IF ANSWER$="YES" THEN 30
110 IF ANSWER$="NO" THEN END "Wasn't that fun?"
120 PRINT "Please answer YES or NO."
130 GO TO 80
140 END
```

```
10 REM FOUR LETTER WORD DETECTOR
20 DIM S$(10)
30 PRINT "I detect four letter words."
40 PRINT "Try typing in a word"
50 INPUT S$
60 IF LEN (S$)=4 THEN 90
70 PRINT "That's fine.  Feed me another."
80 GO TO 40
90 PRINT "You typed a four letter word!"
100 FOR I=1 to 100
110 BEEP
120 NEXT I
130 END "I QUIT !!!"
```

16.5.7 The VAL Function

Form: VAL (<string expression>)

Examples:
```
X=VAL (A$)
PRINT VAL (STR$)
```

The VAL function accepts a string argument. If the string consists of a valid numeric representation (a series of characters representing a numeric constant), the numeric value represented by the string is returned; otherwise, a value of zero is returned. For example:

```
10 DIM S$(5)
20 S$="47.5"
30 PRINT VAL (S$)
40 S$="???"
50 PRINT VAL (S$)
```

will display:

```
47.5
0
```

This function enables the user to write a program which can accept text containing numeric representations, isolate these numerics, and directly convert them for computational use.

CHAPTER 17. FILE INPUT AND OUTPUT

Chapters 11 and 12 discussed input from the keyboard and output to the display. BASICPLUS can also accept input from and deliver output to disk files. Sequential Access, Direct Access, or ISAM disk files created by programs written in other Datapoint languages (space compressed or not) are acceptable to BASICPLUS. For a more detailed explanation of the Datapoint disk file structure, consult the DOS User's Guide.

17.1 Disk File Structure

The smallest unit of disk storage is the sector; all disk I/O (input/output) hardware instructions operate on entire sectors. Each sector can hold 256 bytes (characters) of which 5 are pre-empted by DOS for system use, leaving 251 available.

A group of sectors is called a _file_. Each file has a name associated with it. DOS provides the ability to manipulate files by name. On a system with more than one disk drive, the _drive_-_number_ or _disk_ _volume_ _identifier_ can also be used a part of a file name.

17.1.1 Record Structure

Although the smallest _physical_ unit of disk storage is the sector, there are also _logical_ units involved in disk files. BASICPLUS file I/O deals in both physical and logical records.

A physical record corresponds to exactly one sector on the disk. It starts with the first character of the sector and ends with an octal character 003. Thus, for compatiblity with DOS, there are at most 250 usable data characters in a physical record.

A logical record is a series of characters terminated by an octal 015, therefore a logical record within a single physical record can contain only 249 data characters.

However, logical records are superimposed on physical records: several logical records may be stored in one physical record, or a single logical record may span two physical records.

Example: Four logical records could appear on the disk as:

```
asc asc asc asc asc asc oct asc asc asc asc asc asc oct asc oct
 L   I   N   E         1 015  L   I   N   E         2 015  L  003

asc asc asc asc asc oct asc asc asc asc asc asc oct oct
 I   N   E         3 015  L   I   N   E         4 015 003
```

Note that the first physical record contains two logical records as well as the first letter of a third. The octal 003 ends the physical record; the remaining 235 characters of the sector are ignored by BASICPLUS. The third logical record starts in the first physical record and continues into the second physical record. At this point the fourth logical record starts and continues to the end of the physical record.

If the same four logical records were written to the disk, one per physical record, they would appear as:

```
asc asc asc asc asc asc oct oct
 L   I   N   E         1 015 003

asc asc asc asc asc asc oct oct
 L   I   N   E         2 015 003

asc asc asc asc asc asc oct oct
 L   1   N   E         3 015 003

asc asc asc asc asc asc oct oct
 L   I   N   E         4 015 003
```

Note that the first example took 2 sectors to store 4 logical records, while this second example took 4 sectors to store the same amount of information. The latter style of disk formatting is sometimes preferred over the former style, because the user may be able to gain faster and easier access to the logical record of his choice at the expense of storage space.

17.1.2 End-of-File Mark

The end-of-file mark (EOF) is a special type of physical record which is written to the disk as the last physical record of a file. It always starts at the beginning of a physical record and has the following format:

```
oct oct oct oct oct oct oct
000 000 000 000 000 000 003
```

The rest of the characters in the sector are of no significance.

All records between the beginning of the file and the EOF must be in acceptable physical record format. Any record that is not in this format will cause an I/O or FORMAT trap. An empty file is acceptable; that is, any file which has an EOF as its first physical record is acceptable.

17.2 File Accessing Methods

All disk I/O in BASICPLUS is based on establishing a position within a file. Once this position is established, all accesses are performed by moving this position within the file. This position within the file is completely described by two values maintained by BASICPLUS: the _record number_ and the conceptual _character pointer_.

The record number specifies which sector is currently being referenced. The value zero (0) specifies the first sector.

The character pointer specifies the byte currently being referenced within the sector. The value one (1) specifies the first byte of the sector. The only control the user has over this character pointer in BASICPLUS is that each character printed out and each TAB function reference advances the pointer along the record, and a RESTORE sets the pointer to the first character of the record to which the RESTORE refers.

17.2.1 Physical Record Access (Direct Access)

Physical record access is the fastest and simplest method of accessing information within a file. Physical record accessing may be used to randomly access information on the disk.

Each physical record in a file is associated with a positive integer value starting with zero (0). To access a given physical record, the user must specify the record number of the physical record desired. The position in the file is altered such that the record number of the file is set to the specified value and the character pointer is set to the value one (1). Once the position has been established, the access continues as if it had been a logical record access.

17.2.2 Logical Record Access (Sequential Access)

This is the access method used to read and write logical records. This access method allows only sequential processing of disk records. If the programmer requires random access to logical records, he must either use a combination of direct access and sequential access or use the slower but more powerful indexed accessing (see below).

In reading a logical record, all characters in the file from the current position of the record number and character pointer up to (but not including) the next octal 015 are read.

17.2.3 Indexed Sequential Record Access (ISAM)

An indexed sequential file is a sequential file in which each record contains a _key_--an alpha-numeric field which uniquely identifies that record. Using ISAM, a program can access records by key rather than by position within the file. All keys must be the same length and each must be located at the same offset from the beginning of its record.

An indexed sequential file alone is indistinguishable from an ordinary sequential file. What makes it _indexed_ sequential is the presence of another file containing an index to the sequential file.

The index file contains the name and extension of the file which it indexes, copies of the keys, and the pointers necessary to associate the keys with the logical records of the indexed file in a sorted order. Index files may be created only by the DOS utility "INDEX", which is described in the DOS User's Guide. There _may not_ be more than one index file associated with a single indexed sequential file. The keys in an index file are stripped of trailing spaces, since unnecessary spaces cause larger index files and longer access times.

Since ISAM files are handled quite differently from Direct Access and Sequential Access files, they are described separately below in section 17.6.

17.3 General File I/O Operations

BASICPLUS I/O statements specify files by device number (1-3 for disk, 4 for printer). DOS manipulates files by name. The OPEN or TOPEN statement is used to associate a device number with a named DOS disk file. (The printer is always associated with device number 4; No OPEN is used.) The device number is then used to specify the file in PRINT, INPUT, LIST, CAT, FREE, NOFILE, RESTORE, VARS, and END statements.

File #3 should never be in use when a SAVE, ROLLOUT, SCRATCH, GET, or APP command is executed, since these commands "close" file #3.

A handy way of specifying an output device is to use a variable. Suppose Q is used throughout the program to denote the output device. Then during debugging, Q can be set to zero to direct output to the display; and during production, Q can be set to the numeric value of the proper output device.

17.3.1 The OPEN and TOPEN Statements

```
Form:       OPEN # <numeric expression> , <string expression>
            TOPEN # <numeric expression> , <string expression>

Examples:   OPEN #2, "PAYMENTS/MST:DR0"
            OPEN #N, A$
            TOPEN#3,"DATAFILE"
            OPEN #1, P$ + ":DR" + ( "#" USING D )
```

The string specified in the OPEN or TOPEN statement gives the file-name, file-extension and drive-number or disk volume name. If no file-extension is specified, the default extension is "/TXT". If no drive-number or disk volume name is given, all drives on line are searched for a file with the specified name and extension. The search commences at the lowest numbered drive and continues in ascending numeric order. The first one found is opened. If the OPEN statement is used, and the file is not found, it will be created and opened on the lowest-numbered available drive which contains room in its directory and enough space to allocate at least one segment for the file (This size depends on the DOS in use, and varies from three to 24 sectors). If the TOPEN statement is used and the file does not exist, it will not be created, and no file will be opened. The success of the TOPEN statement may be tested with the NOFILE condition (see below).

Once a disk file has been opened and used for input, it is in "read status". It can be used for INPUTs indefinitely. When the end-of-file mark is encountered, the EOF condition becomes true for that file, the values returned from the INPUT are zeros or null strings, as appropriate, and the file is placed in "write mode" (i.e., it may be PRINTed on). Any INPUT performed after the EOF condition becomes true will cause a "NO MORE DATA" message, and the values of the variables of the input list will remain unchanged.

17.3.2 The END# Statement

Form: END # <numeric expression>

Example: END#2

The END statement with an I/O device number causes the specified disk file to be "closed". The disk file named in the OPEN statement for the specified device number is disassociated from the device number, releasing the number for reassignment. If the file was a sequential output file, an end-of-file mark is written after the last data record. This is useful for changing a unit from reading to writing, or vice-versa.

For example, assume that an intermediate disk file "WORK1/TXT" has been written, and it should next be read. It must be closed and re-opened for reading. END#2 will close the disk file, and OPEN#2,"WORK1" will reopen the file allowing it to be read from the beginning without stopping the program.

END with #<numeric expression> does not stop execution. END without #<numeric expression> implies all devices and in addition, stops execution. The END#<numeric expression> is particularly useful if the BASIC program is run under chaining with DOS. The

DOS command (see section 20.1.6) can follow an END#<numeric expression> but not END alone.

17.3.3 The EOF Condition

```
Form:      EOF # <numeric expression>

Examples:  IF EOF#1 THEN 17
           IF EOF#0 THEN PRINT "END OF DATA."
```

This condition is used to test whether an INPUT (or a READ, in the case of #0) has encountered an end-of-file mark. When the program attempts to read beyond the end-of-file mark, the "NO MORE DATA" message is displayed and the values of the variables in the input list remain unchanged. The EOF condition may be used in conditionals to cause a specified operation if end-of-file has been encountered. EOF#0 (or just EOF) is used to detect the end of the DATA sequence (see chapter 13).

The EOF condition can be used to position to the end of a file so that new data can be added. This is done by INPUTting until EOF becomes true, at which point BASICPLUS places the file in "write mode", and the PRINT command can be used to append the new data.

17.3.4 The NOFILE Condition

```
Form:      NOFILE # <numeric expression>

Examples:  IF NOFILE#1 THEN 17
           IF NOFILE#2 THEN PRINT "NO SUCH FILE!"
```

The NOFILE condition is used to test whether or not a specific file number is currently open. If #0 is used, TRUE will always be returned. NOFILE#4 will return FALSE only if the printer is available for exclusive use by BASICPLUS, and has not been released (This is true from the time a PRINT#4 is executed, until some type of file-closing statement is executed). For all other files, it returns TRUE if the file is closed, or FALSE if the file number is currently open. This statement has particular value in testing the success of a TOPEN statement:

```
10 TOPEN#1,"DATAFILE"
20 IF NOFILE#1 THEN PRINT "DATA FILE MISSING!";STOP
30 PRINT "DATA FILE HAS BEEN OPENED."
```

17.3.5 The SIZ Operator

Form: SIZ # <numeric expression>

Examples:
```
SIZ#0
SIZ#3
SIZ#1
A=INT(SIZ#1/2)
```

The SIZ # operator is used to determine the width of the specified output device. The size of disk files is normally 249. The size of the CRT display will normally be returned as 80. The size for printers varies depending on the size with which the printer was configured (typically 132 or 79). If no number sign and device number are specified, #0 is assumed. The value returned by the SIZ operator may be used in any numeric expression, as in the fourth example.

17.4 Sequential File I/O

17.4.1 The File PRINT Statement

Form: PRINT # <numeric expression> , <print list>

Examples:
```
PRINT #3,VAL1,VAL2,VAL3
PRINT #F,PAYMENT
```

The file PRINT statement is used to write a logical record to the processor CRT screen (#0), a disk file (#1-#3), or the printer (#4, see chapter 16). The separators in the print list may be commas or semicolons. The separators have the same effects that they have in the ordinary PRINT statement (see chapter 11): commas cause zoned output and semicolons cause directly appended output.

A separator following the last item in the print list indicates that more data will be added to the display or print line or disk file logical record by another PRINT statement. In the case of disk files, this means that no octal 015 (end of logical record) will be written. If an item is to be written into a file and there is not enough space remaining in the sector to write the entire item, as much of the item as will fit is written followed by an octal 003 to mark the end of the physical record. The remainder of the item is then written at the beginning of the next sector. The END # and RESTORE # statements also cause an octal 003 (ETX character) to be written after the last character written in the current record.

17.4.2 The File INPUT Statement

```
Form:       INPUT # <numeric expression> , <input list>

Examples:   INPUT #2,A,B,C
            INPUT #I,FORCE,TIME
```

The file INPUT statement is used to input a logical record from the keyboard (#0) or disk file (#1-#3). Input from the printer (#4) is, of course, illegal. One line of input from the keyboard or one logical disk record corresponds to one INPUT statement. If there is less data available then there are corresponding variables in the INPUT statement, the remaining variables will be set to zero. If there is too much data in the line or record, the excess will be ignored. When a disk file INPUT statement has been completed, the character pointer associated with that file is advanced to just beyond the ending octal 015.

17.4.3 Example of Sequential File I/O

The following program reads a source file from the disk and prints it on device Q.

```
10 PRINT "List a source file on device Q"
20 DIM S$(80)
30 PRINT "Enter File Name: ";
40 INPUT S$
50 TOPEN#2,S$
60 IF NOFILE#2 THEN PRINT "No such name!";BEEP;GOTO 30
70 N=0
80 PRINT "List on Display or Printer? ";
90 Q=-1
100 INPUT S$
110 IF S$="D" THEN Q=0
120 IF S$="P" THEN Q=4
130 IF Q<0 THEN 70
140 INPUT#2,S$
150 IF EOF#2 THEN 190
160 N=N+1
170 PRINT#Q,S$
180 GO TO 130
190 PRINT "End of file after ";N;" records."
200 END
```

17.5 Direct Access File I/O

17.5.1 The RESTORE # Statement

Form: RESTORE # <numeric expr> [, <numeric expr>]

```
Examples:  RESTORE#2
           RESTORE#2,37
           RESTORE#1,A
```

This form of the RESTORE statement is used to provide direct access capability for processing disk files. The first number specified is the device number used in the OPEN statement. The second number is optional and is used to position the file pointer to the first character of a specific disk file _physical_ record (sector). If the second number is omitted, the file pointer will be positioned to the first character of physical record 0 (the first record of the file).

To write a simple direct access file, precede each PRINT statement with a RESTORE # statement. This will write one logical record beginning at the first character of the specified physical record, and continuing for as many physical records as necessary. A subsequent RESTORE # statement will insert an octal 003 (end-of-physical-record) after the last character written in the current record, write the record to disk, then position to the first character of the physical record (sector) specified.

To input records from a simple direct access file, precede each INPUT statement with a RESTORE # statement specifying the record number. The INPUT will then fetch one logical record beginning at the first character of the specified physical record.

When direct access files are closed by the END statement, BASICPLUS does not write an end-of-file mark. If an end-of-file mark is required, the user must write a 6 character string containing all binary zeros after the last data record and then END the file. For example:

```
100 DIM A$(6)
110 A$(1)=A$(2)=A$(3)=A$(4)=A$(5)=A$(6)=0
120 PRINT#1,A$;
130 END#1
```

The semicolon is used to prevent output of a (015) in statement 120.

17.6 ISAM Files

BASICPLUS has the ability to manipulate ISAM (Indexed Sequential Access Method) files. The records of an ISAM file are organized on the basis of a collating sequence determined by a specific "key" within each record. The key is a portion of the record which uniquely identifies that record from all others in the file. Within one file, all keys must have the same length and each must be located at the same position of its record.

The ISAM file actually consists of two files, an index file and an ordinary sequential text file. The index file is used to locate the text file record which contains a given, unique key.

There <u>may not</u> be more than one index file associated with a single indexed sequential file.

The text file may be created by any of the standard text editors (i.e., EDIT, BASICPLUS, BASIC55, DOSBASIC, DATABUS, DATAFORM, and so on). The index file is created by using the INDEX utility (see the DOS User's Guide). Using this index file, BASICPLUS can index to the record associated with a specific key, update records, delete and insert records, and perform several other ISAM functions.

17.6.1 The IOPEN Statement

```
Form:       IOPEN # <numeric expression> , <string expression>

Examples:   IOPEN #1,"SCRATCH"
            IOPEN #3,A$
```

The IOPEN statement is used to open an index file. Opening the index file also causes the indexed file to be opened implicitly. Since the indexed file does not use a device number, a program may OPEN files #2 and #3 as ordinary disk files and still IOPEN #1 as an ISAM file, since the indexed file itself does not use a device number.

The default extension for file names is "ISI". If the specified index file does not exist, the error message "FILE DOES NOT EXIST" will be issued. An index file must be opened before any other ISAM file commands are given. If the format of the index file is not correct, the message "BAD ISAM FILE" will be displayed, and the INDEX utility should be run (See DOS User's Guide).

17.6.2 The RESTORE by Key Statement

```
Form:       RESTORE # <numeric expr> , <string expr>

Examples:   RESTORE #1,"HELLO"
            RESTORE #2,KEY$
            RESTORE #N,""
```

This special form of the RESTORE statement is used to position the specified ISAM file to a specific key. If the file specified was not previously opened in an IOPEN statement, the error message "FILE NOT ISAM" will be issued. The program may return to the beginning of the index file by specifying a null key, as in the third example. If the specified key can not be found, the file will be positioned to the next key in the sorted index.

Care must be taken in using the RESTORE by Key statement. Suppose the user wishes to locate and update the record associated with a certain key. The user must RESTORE by key to locate the record and INPUT to get the record. The record can then be inspected and modified. The INPUT, however, advances the character pointer to the next logical record just as it would in an ordinary sequential file INPUT. To correct for this, the user must <u>RESTORE by key again</u> to back up the character pointer to the beginning of the record, and <u>then</u> UPDATE to store the new version into the file.

17.6.3 The NEXTKEY Statement

```
Form:       NEXTKEY # <numeric expression>

Examples:   NEXTKEY #(A+1)
            NEXTKEY #3
```

The NEXTKEY statement is used to position an ISAM file to the next sequential key. If the file specified was not opened in an IOPEN statement, the message "FILE NOT ISAM" will be displayed.

17.6.4 The NOKEY Condition

```
Form:       NOKEY # <numeric expression>

Examples:   IF NOKEY #1 THEN END; ELSE 10
            IF NOKEY #A THEN BEEP; GO 80
```

The NOKEY condition is used to detect that an ISAM file has been positioned to a non-existent key. This condition works the same way as the EOF condition, except that it tests if the file was positioned by a key. For example:

```
10 DIM A$(80)
20 IOPEN#1,"NAMES"
30 PRINT "FIND: ";
40 INPUT A$
50 RESTORE#1,A$
60 IF NOKEY#1 THEN 90
70 INPUT#1,A$
80 GO 30
90 PRINT "KEY NOT FOUND."
100 GO 30
```

would display user specified data from the index file "NAMES".

17.6.5 The INSERT Statement

Form: INSERT # <numeric expression> , <string expression>

Examples:
```
INSERT #2,"NAME"
INSERT #3,INS$
```

The INSERT statement is used to insert a new key in the list of keys. The new key will point to where the next data line will be written upon the execution of the next PRINT statement. If the file referenced is not an ISAM file, then "FILE NOT ISAM" will be displayed. If the specified key already exists, the message "KEY ALREADY EXISTS" will be displayed and no insertion will be done.

17.6.6 The UPDATE Statement

Form: UPDATE # <num expr> [, [<expr> <sep>]... <expr>]

Examples:
```
UPDATE #2,"NAME: ";NAME$
UPDATE #Q,"EMPLOYEE #: ";EMP
```

The UPDATE statement works like the PRINT statement, except that instead of appending to the file, it prints in place. That is, the record that the file is currently positioned to will be overwritten by whatever is printed via the UPDATE statement. Note that the length of the data which is updated must be exactly the same as the length of the data which was previously in the file (i.e., a line which is 67 characters long may not be updated over a line which was previously 40 characters long).

17.6.7 The DELETE Statement

```
Form:       DELETE # <numeric expression>

Examples:   DELETE #1
            DELETE #Q
```

The DELETE statement deletes the key and the text that the key points to, after which a NEXTKEY is performed. The key and text will then be ignored by subsequent text handling statements and ISAM statements. After a time, the file may become cluttered and quite large if many deleted records are present. Use REFORMAT (See DOS User's Guide) to reorganize the file.

17.6.8 The Structure of Index Files

This section is included as a reference for those users who want more insight into the structure of ISAM index files. Mastery of this section is not a prerequisite to the successful use of ISAM files.

The index structure is an N-ary tree. N is determined by the number of keys that will fit within a disk sector. Each node of the tree is contained within one disk sector. The tree has enough levels so that the uppermost node will fit within one disk sector.

The lowest level of the tree is a linked list. The keys in the linked list are arranged sequentially according to their ASCII values.

Depending on the length and path of this linked list, the time spent in traversing this list can lead to considerable overhead. The INDEX utility may be used to reorganize this list to minimize the time spent in traversing it. USE THE INDEX UTILITY FREQUENTLY!

The simplified diagram below demonstrates the manner in which the keys are associated with the logical records. The diagram assumes that only 3 keys will fit per sector and that the data file was indexed on column 6. The upper half models the index file; the lower half, the indexed file. The character "*" denotes a pointer. The character "0" denotes a null pointer. The character "$" denotes a non-existent key value. Sector boundaries are denoted by "!".

```
      ! A * J * $ 0 !
         /     \
        /       \
       /         \
      /           \
     /             \
     |             |
     |             |
     v             v
   ! A * D * G * ! J * $ 0 $ 0 !
       |   |   |     |
      /     \   \     \__________________________________
     /       \   \____________________                   \
    /         \____                   \                   \
   /               \                   \                   \
  |                 \                   \                   \
  |  ____  ____  ____\_  ____  ____  ____\_  ____  ____  ____\_
  | |   | |   | |     | |   | |   | |     | |   | |   | |     |
  v |   v |   v |     v |   v |   v |     v |   v |   v |     v
! A * * B * * C * * ! D * * E * * F * * ! G * * H * * I * * ! J * $ 0 ...!
      |     |     |       |     |     |       |     |     |     |
      |     |     |       |     |     |       |     |     |     | index file
----------------------------------------------------------------------------
----------------------------------------------------------------------------
      |     |     |       |     |     |       |     |     |     | indexed fil
      |     |     |       |     |     |       |     |     |     v
      |     |     |       |     |     |       |     |     v    LINE J.
      |     |     |       |     |     |       |     v    LINE I.
      |     |     |       |     |     |       v    LINE H.
      |     |     |       |     |     v      LINE G.
      |     |     |       |     v    LINE F.
      |     |     |       v    LINE E.
      |     |     v      LINE D.
      |     v    LINE C.
      v    LINE B.
     LINE A.
```

17.7 File Hints

The files used by Datapoint BASICPLUS are written so that they are compatible with the DOS and CTOS editors. Therefore, data and programs can be prepared using DOS or CTOS editors (transfer to and from disk with MIN and MOUT). Files in editor format are also accepted by all other Datapoint software systems. Numbers and strings are kept in the same format--as edited characters. Therefore, output can be written as strings and read back as numbers and vice versa.

Input and output can be formatted using BASIC strings since the strings are of constant length. For example, if input has first name in columns 1-10 and last name in 11-20, the following will handle it:

```
10 DIM FIRSTNAME$(10), LASTNAME$(10)
20 INPUT#1, FIRSTNAME$, LASTNAME$
```

Numeric and string fields can be mixed. The numeric field will end with the first character that cannot belong in the number. The string field will end after getting enough characters to fill the string.

To get right-justified, columnar output use formatted printing (see section 16.5.3, String Editing).

CHAPTER 18. PRINTER OUTPUT

BASICPLUS allows the user to print output directly to a printer. Local, remote, and servo printers are accessible in this manner. Chapter 2 describes how to configure any available printer at the beginning of a BASICPLUS session.

Just as the disk files are accessed by device numbers 1 through 3, and the display can be accessed by device number 0, the printer is always accessed by device number 4.

18.1 The @ Function

The dummy function "@" will return, at any time, a value which informs the program (or the user) which printer has been enabled. The values for the different printers are:

0	No printer enabled
4.1	Remote printer enabled
4.2	Local printer enabled
4.3	Servo printer enabled
4.4	Microplotting servo printer enabled

Note that the use of a fractional part for the representation allows the user to use PRINT#@ (since the value of the @ is rounded back to 4), and also allows the program to test which printer has been enabled. This would be useful in a program which uses microplotting, since it could inform the user if it had not been properly invoked with the "M" option.

The @ may also be used anywhere a numeric expression can be used, for example:

```
LET A = @
```

18.2 General Remarks on Printer Usage

Unlike disk files, the printer cannot be explicitly OPENed. The initiation of a BASICPLUS session will open the printer that is configured.

The printer cannot be closed (such as with an END #4 statement).

The printer never reaches end-of-file, so the testing of EOF #4 is not valid.

The SIZ #4 operation will return the printer record size.

The PRINT #4 statement will send its output to the printer. The comma and semicolon separators have the same significance that they have on the display. Commas provide left-justified columnar output; semicolons provide close-packed output.

The trailing comma and semicolon on PRINT #4 statements also have the same significance that they have on the display: the next PRINT #4 will continue where the current one ended.

18.3 Formatting Printer Output

Special functions in a PRINT#4 statement allow paging and overprinting on the printer:

LF - Line Feed
FF - Form Feed (page eject)
CR - Carriage return and suppress line feed

These functions are the same as the display formatting functions, except that they may only be output to the printer. These functions must also be followed by a comma if more print items follow on the same line. If used on the end of the line, they may be followed by either a semicolon or nothing, depending on the formatted output desired. For example, the following will overprint one line and page eject:

```
10 PRINT#4,"ABCDEFGH";CR,"012345676";FF
20 END
```

The older method of the print formatting instructions is still valid. In using this method, special code numbers in strings allow paging and overprinting:

```
10 - Line Feed
12 - Page Eject
14 - Carriage return and suppress line feed
```

The following program is equivalent to the above example:

```
10 DIM CR$(1),PG$(1)
20 CR$(1)=14;PG$(1)=12
30 PRINT #4,"ABCDEFGH";CR$;"01234567";PG$
40 END
```

18.4 Plotting With The Servo Printer

The servo printer can be micro positioned for plotting by using the special co-ordinate positioning feature of BASICPLUS. The micro co-ordinate feature allows direct positioning to any of 589,824 micro positions on a page before printing. To use this feature a ";M" must be entered in the command line when loading BASIC.

Type "BASICPLS;M" and the message "MICRO PLOTTING ENABLED" will appear during initialization if your processor has a servo printer attached and ready. If the feature is not selected, the memory used to support micro plotting will be added to the users work space.

Micro positioning co-ordinates are defined as follows:

```
10 DIM A$(5)
20 A$(1)=15
30 A$(2)=HORIZ/256
40 A$(3)=HORIZ
50 A$(4)=VERT/256
60 A$(5)=VERT
```

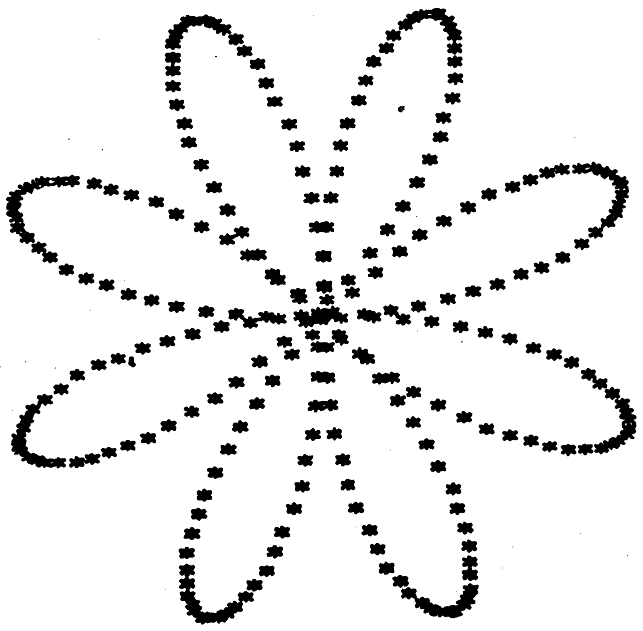
Statement 10 defines a five character string that will direct the servo printer to position to a micro co-ordinate. Horizontal and vertical co-ordinates must be in the range 0 to +768. A$(1) contains the special function code 15 that indicates to the servo printer driver that co-ordinates follow. A$(2) and A$(3) contain the horizontal co-ordinate. A$(4) and A$(5) contain the vertical co-ordinate. Statement 30 causes BASIC to take the floating point variable "HORIZ" and divide it by 256, convert the result to an integer and store the result in A$(2). Statement 40 causes BASIC to take the same variable and convert it to an integer and store the least significant byte (value 0-255) in A$(3). Statements 50 and 60 store the vertical co-ordinate.

Micro positioning moves the print carriage and printer platen simultaneously along the most direct path to reach the desired position. Horizontal co-ordinate 0, vertical co-ordinate 0 is the position of the carriage and platen following the last carriage return executed. When BASICPLUS is loaded with the M option, a carriage return is done immediately. Since 768 vertical micropositions equals some 16 inches, and the coordinate origin is "bottom left", remember to advance the paper at least 2 sheets before beginning a plot.

When generating co-ordinates, consideration should be given to the fact that five horizontal micro increments equal four vertical micro increments on a servo printer. The printer can position to 60 micro positions per inch horizontally and 48 micro positions per inch vertically. For example:

```
10 DIM S$(5)
20 S$(1)=15;TWOPI=6.28319;I=2.32711E-2
30 FOR A=I TO TWOPI STEP I
40 X=SIN(4*A)
50 H=160+100*COS(A)*X;V=80+80*SIN(A)*X
60 S$(2)=H/256;S$(3)=H
70 S$(4)=V/256;S$(5)=V
80 PRINT #4,S$;"*";
90 NEXT A
100 END
```

will produce this star-like pattern:



IMPORTANT!

Co-ordinate positioning strings must not be output to any device except the servo printer or the result will be unpredictable. All five elements of the string must be output as a single element or loss of sychronization will occur. Note the semicolon terminating statement 80 to prevent the printer from doing a carriage return after printing!

Note: The 1500 does not have the capability to drive a servo printer, and micro-positioning should not be attempted on this processor.

CHAPTER 19. PROGRAM EDITING

BASICPLUS provides editing commands similar to those in EDIT to allow modification and deletion of program statements.

19.1 Statement Modification

```
Form:       :M <line number> <old text> <modifier> [ <new text> ]

Examples:   :M 120 ABCD<XYZ
            :M 20 20<25
            :M 80 1234>567890
            :M 80 12|567890
            :M 20 ABC<
```

Modifiers:

<	Replace first occurrence of <old text> with <new text>
>	Append <new text> to first occurrence of <old text>
|	Concatenate <new text> to first occurrence of <old text>
\	Concatenate <new text> to first occurrence of <old text>

The :M command is used to modify statements.

In the first example, the characters "ABCD" in statement 120 are replaced by the characters "XYZ" and statement is re-entered.

In the second example, the statement number portion of statement 20 is changed to "25" and a new copy of the statement is entered as statement 25. If statement 25 already exists then it will be replaced. The original statement 20 is _not_ deleted.

In the third example, the characters "567890" are inserted following the characters "1234" in statement 80 and the statement is re-entered.

In the fourth example, the characters "567890" are appended to the characters "12" in statement 80. Any characters that followed "12" previously are deleted. The modified statement is re-entered.

In the last example, the characters "ABC" in statement 20 are deleted (old-text replaced by no new-text) and the statement is re-entered.

Notice that in the examples a new statement is created from an existing statement in exactly the same way it would be entered from the keyboard. If the statement number is changed, the statement will be entered as a new statement and the old statement will still be present and unchanged. If all of the characters following the statement number are deleted, the statement will be deleted. If characters to the right of the statement number are changed, the new statement will replace the old statement.

The :M command can only be entered from the keyboard! It can not be embedded in programs read from the disk. Exactly one space must precede and exactly one space must follow the line number for correct interpretation by the BASICPLUS editor.

19.2 Statement Deletion

```
Form:       :D <line number> [ - <line number> ]

Examples:   :D 125
            :D 30-100
```

The :D command is used to delete statements. In the first example, statement 125 is deleted (if it exists). In the second example, all statements with line numbers between 30 and 100 inclusive are deleted.

Exactly one space must precede the first line number.

The :D command can be entered from the keyboard or embedded in programs and program overlays stored on disk. This is used in conjunction with program chaining (see chapter 21). This command will not produce a diagnostic if a matching line number is not found.

CHAPTER 20. UTILITY COMMANDS

This group of commands enables the user to manipulate and investigate the BASIC program in memory, identify the files stored in the DOS disk directory, and maintain and access program libraries. Some of the commands may also be used as statements in a stored BASICPLUS program.

20.1 Program Manipulation and Termination

20.1.1 The RUN Command

```
Form:       RUN [ <line number> ]

Examples:   RUN
            RUN 40
```

The RUN command is used to initialize BASICPLUS and execute the stored program. If a line number is supplied, execution begins with that line number.

Specifically, the initialization steps performed include:

- Write any incomplete output on disk
- Close any open disk files
- Clear any arrays previously in use
- Clear any memory of previous GOSUBs
- Clear any memory of previous FOR loops
- Clear any memory of previous defined functions
- Cause any READ statements to read from the first DATA statement
- Reset the RANDOM NUMBER generator to the first value

Note that simple variables retain their values and are NOT reset to zero or undefined. This enables the user to run one program computing certain values, and then chain to another program passing the computed values along (see chapter 21, Program Chaining).

20.1.2 The RENumber Command

Form: REN [<line number>] [, <incr>]

Examples:
```
REN
REN 5
REN 10,10
```

The REN command renumbers the entire program currently in memory, changing both the line numbers at the beginning of each line and the references to line numbers in control statements. This command is useful when the program is being modified and it becomes necessary to insert a block of lines between two lines whose line numbers are too close together to permit that. The command is also useful to enhance the appearance of completed programs. The REN command may not be used in a program. It is only to be entered from the COMMAND mode, and it cannot be embedded in programs read from the disk.

The program is renumbered to start with a line number of <line number> and incremented by <incr>. These values both default to 10. For example, if the following program were typed in:

```
10 PRINT "HELLO THERE."
20 GO TO 47
27 GOSUB 99
32 STOP
47 PRINT "HOW ARE YOU?"
54 GO 27
95 END
```

If the command "REN 5,5" were entered, BASICPLUS would renumber the program starting with a line number of 5, and incrementing by fives:

```
5 PRINT "HELLO THERE."
10 GO TO 25
15 GOSUB 99
20 STOP
25 PRINT "HOW ARE YOU?"
30 GO 15
35 END
```

Note that the GO TOs and GO SUBs were also changed to point to the appropriate lines. Also note that line 15 with a GOSUB to line number 99 (which is a non-existant line) was not changed.

The KEYBOARD key may be used to stop REN at any time. When a given line's line number is changed, all references to that line number are also changed. Therefore all references to line numbers which have been resequenced, and all references to line numbers which have not, are correctly handled. The program may then be resequenced with another REN command.

Errors which may occur during renumbering are:

LINE NUMBER > 38399

> This means that in renumbering the program, a line number of greater than 38399 was created. Renumber the program again using a smaller <line number> and/or <incr>.

NO ROOM

> It is possible that a program may become larger during renumbering if a GOTO line number becomes greater in length (i.e., GOTO 100 changes to GOTO 2000). The only way to solve this is to RENumber the program again using a smaller <line number> and/or <incr> so as to assure a GOTO line number of remaining the same length. REN 1,1 will always work on an existing program.

It is possible for renumbering to lengthen a line of text more than seventy-nine characters, such as when an ON GOTO with several two-digit line numbers is changed to have several four-digit line numbers. _BASICPLUS cannot warn the user about this_. If such a line is SAVEd and then fetched with a GET, text will have been lost.

20.1.3 The AUTO Command

Form: AUTO [<line number>] [, <incr>]

Examples:
```
AUTO
AUTO 5,5
```

The AUTO command causes BASICPLUS to automatically generate line numbers, allowing the user to type in the statements on the pre-numbered lines. The line numbers start at <line number> and are incremented by <incr>. Both <line number> and <incr> default to 10 if not specified. AUTO will terminate if a line number is created which already exists, a line number greater than 38399 is created, or a null line is entered. The AUTO command may not be used in a program. It is only to be entered in the COMMAND mode, and it connot be embedded in programs read from the disk.

20.1.4 The ERASE Command

```
Form:       ERASE [ <variable> [ , <variable> ]... ]

Examples:   ERASE
            ERASE A,B,C
            ERASE VAR1,VAR2,NEXTVAR
```

The ERASE command resets a variable to the undefined condition. It can be used to erase the contents of single variables (as in the second and third examples), or to erase the contents of all variables currently in memory (as in the first example). This command <u>does not</u> release the memory space allocated to the variable.

Note: erasing an array does not release the space allocated to that array.

20.1.5 The SCRATCH Command

```
Form:       SCRATCH

Example:    SCRATCH
```

The SCRATCH command causes everything BASICPLUS knows to be erased. If disk files are being accessed, file processing is terminated and the files are "closed". The workspace is cleared. The stored program is erased. All variables are erased, their names forgotten, and their allocated space released. This command is useful to get a clean workspace for new work. See the "Program Libraries" section below for another form of SCRATCH.

20.1.6 The DOS Command

```
Form:       DOS [ <string expr> ]

Examples:   DOS
            DOS "EDIT PROGRAMX/ABC"
```

The DOS command forces input or output from/to disk to be completed, closes the files and exits to the Disk Operating System. If a string is specified, then when the Disk Operating System is reloaded, the program specified will be executed.

20.2 Program Investigation

20.2.1 The LIST Command

```
Form:       LIST [ # <numeric expr> , ] [ <line number> ]

Examples:   LIST
            LIST #4, 120
```

The LIST command lists a copy of the current stored statements on the device specified by the numeric expression. If no device is specified, the display is assumed.

If <line number> is included, the listing begins at that line. If the <line number> does not exist, then the listing will begin at the line with the next line number greater than that specified. If the <line number> is greater than the last line number of the program, BASICPLUS will return to command mode ("READY").

Use the DISPLAY key to hold output on the screen. Use KEYBOARD to cut a LISTing short.

20.2.2 The FREE Command

```
Form:       FREE [ # <numeric expr> ]

Examples:   FREE
            FREE #4
```

The FREE command lists the remaining program space on the device specified. The first number listed is the number of bytes of program space remaining and the second number is the number of dictionary name entries available.

20.2.3 The CATalog Command

```
Form:       CAT [ # <numeric expr> , ] [ <string expr> ]

Examples:   CAT
            CAT "ORBIT/PLT"
            CAT #4, "A:DR2"
```

The CAT command selectively lists the names of files found in the DOS disk directory on the device specified. If no device is specified, the display (device #0) is assumed. If the string expression is present to specify a name (possibly qualified by extension, drive-number, or disk volume identifier), then the print out will contain only the specified names.

A completely qualified file name consists of three parts: file name (8 characters), extension (slash followed by 3 characters), and drive specifier (colon followed by either "DR" <digit> <digit> to specify the drive-number or by up to eight characters to specify the disk volume identifier). If the completely qualified file name is specified in the CAT command, and a file of that name is present in the DOS directory, the CAT command will display the name.

If only a partially qualified file name is specified, then the CAT command will list the names of <u>all</u> files which match the portions of the name which were specified. In fact the file name and extension parts can, themselves, be partially specified. Thus, the command CAT "REF" will display the names of all files which begin with "REF", having any extension, and located on any drive on-line; CAT "/T" will display the names of all files whose extensions begin with "T", on any drive on-line; and CAT "REF/T" will display the names of all files whose names begin with "REF" and have extensions beginning with "T", on any drive on-line. (see the DOS User's Guide).

20.2.4 The VARS Command

Form: VARS [# <numeric expr>]

Examples:
```
VARS
VARS #4
```

The VARS command lists, on the specified device, all variable names currently in use. If no device is specified, the display (device #0) is assumed. The variable names are packed on a line with one space following each name. Variables which are currently undefined are marked with an asterisk (*). Array names and string names are followed by their dimensions enclosed in brackets. If a variable name appears by itself, then it is a scalar variable which is currently defined.

Here is a sample VARS command output:

```
*A ABC XYZ[3,4] A$[1] CC[5]
```

Variable A is an undefined scalar variable. Variable ABC is a defined scalar variable. XYZ is an array with dimensions 3 by 4. A$ is a one-character string array. CC is a one-dimensional array containing 5 elements.

20.3 Program Libraries

BASIC program libraries may be created, accessed, and maintained on disk by using the BASIC commands; SAVE, GET, ROLLOUT, APP and a special form of the SCRATCH command.

20.3.1 The SAVE Command

Form: SAVE [<line number> ,] <string expr>

Examples:
```
SAVE "PROG1"
SAVE "PROGNAME/B11:DR1"
SAVE 9000,"PROG2"
SAVE P$
```

The SAVE command copies the program currently in memory to disk, giving it the name specified by the string expression. The string expression must specify the file-name, file-extension and drive-number or volume identifier. The default extension is "/BAS". If no drive-number or volume identifier is provided, the program will be stored on the first disk that has room (beginning the search with the lowest drive-number on-line). If a line number is specified, then only the portion of the program from the line with that number to the end will be SAVEd.

20.3.2 The GET Command

```
Form:       GET <string expr>

Examples:   GET "PROG1"
            GET "PROGNAME/B11:DR1"
            GET P$
```

To recover a program saved with the SAVE command or created using EDIT, use GET followed by a string expression giving the file-name, file-extension, and drive-number or volume identifier The default extension is "/BAS". If no drive-number or volume identifier is provided, the program will examine each drive on-line, starting with the lowest drive number and continuing in ascending numeric order.

When a GET command is issued, the current stored program and the values and names of all variables will be SCRATCHed. If GET is entered in the COMMAND mode, as the program specified is being read, the line number of the line currently being read is displayed. This information is useful for knowing what the last line number of a program is, or for knowing at which line an error occured if an error message is displayed. After a program is completely read in, the "READY" message is displayed.

GET does not close any open files except file number three.

Sometimes the processor memory can become cluttered with variables and values no longer needed. The space wasted in this manner can be reclaimed by performing a SAVE and a GET on the program.

Complex editing can be performed on the program by SAVEing it and using EDIT on the file produced. Once edited, the program can be reloaded with GET.

20.3.3 The APPend Command

```
Form:       APP <string expr>

Examples:   APP "SEGMENT2"
            APP "SEG3/PRG:DR3"
            APP SEGMENT$
```

The APP command is used to indicate that the contents of a disk file is to be appended to the current program. As in the GET command, if APP is entered in the COMMAND mode, while the program is being APPended the line number of the line currently being read will be displayed. The file specification string is in the same form used in the SAVE and GET commands.

When SAVE, ROLLOUT, GET, SCRATCH and APP commands are executed, processing of any file using file#3 is terminated and the file is "closed" (see chapter 17, File Input and Output). Note that GET and APP end execution of the currently running program and return to Command ("READY") mode.

20.3.4 The ROLLOUT Command

```
Form:       ROLLOUT [<line number>,] <string expr>

Examples:   ROLLOUT "PAYROLL"
            ROLLOUT 100,"SUBROUTN/ABS:SCRATCH"
            ROLLOUT NAME$
```

The ROLLOUT command is similar to the SAVE command, in that it writes the program currently in memory to disk. This command, however, writes a directly loadable object program on disk. The default extension for this file name is "/CMD". This is useful when loading a very large program to reduce the time it takes to perform a "GET". The file will contain a copy of the entire BASICPLUS interpreter, along with the BASIC program that was currently in memory. This filename may then be specified from DOS directly from the command line. When the program is loaded, the screen will be rolled up, and a "RUN" statement is executed. If <line number> was not specified in the ROLLOUT command, the RUN will be to the first line of the program. Otherwise, a "RUN <line number>" is executed.

This command may be executed with or without the overlay version of BASICPLUS. Note, however, that extreme care must be exercised when rolling out when using the overlay version. When the program is reloaded, the environment must be exactly the same as when the ROLLOUT was executed. This is because the program must look for the BASICPLUS library containing the necessary overlays. When the ROLLOUT occurs, BASIC stores the drive number and physical disk location of the library, so that it may be checked upon reload. It is very easy to accidentially modify the physical drive numbers of specific volumes under an ARC environment, since drive numbers become entirely logical. The output file will usually be well over 100 sectors long, so make sure that there is a disk with enough available space to hold the command file (especially on a diskette system!).

20.3.5 The SCRATCH Command for Files

```
Form:        SCRATCH <string expr>

Examples:    SCRATCH "PROGRAM1/BAS:DR2"
             SCRATCH "WORK"
             SCRATCH FILENAME$
```

A special form of the SCRATCH command is used to delete unneeded files from the disk. It is identical in meaning to the KILL command under DOS (see the DOS User's Guide). The SCRATCH command must be followed by a string specifying the file-name. The extension is assumed to be "/BAS" if one is not specified. All drives on-line are searched unless a drive is specified.

When this form of the SCRATCH command is executed from the keyboard, BASICPLUS asks the question:

"DO YOU REALLY WANT TO KILL <string>?"

This gives the user a chance to verify that the file is indeed to be deleted. If the user enters "Y", the file is deleted. Any other response will be ignored and the file is preserved. Note: If the file is protected in any way, BASICPLUS will not ask any questions. It will only display "NO!".

Note: If this form of the SCRATCH command is used within a program (e.g., 10 SCRATCH "PROGNAME"), the question "DO YOU REALLY WANT TO KILL PROGNAME?" will not be asked, and the file, if it exists, will be deleted unconditionally.

CHAPTER 21. PROGRAM CHAINING

Chaining is the term used for the serial execution of a series of programs with no user intervention required. If a BASIC program is too large to be executed at once, it may be possible to break it into smaller parts that run sequentially one after another. If necessary, data can be passed from one program to the other using disk files for intermediate storage.

21.1 Chaining Within BASICPLUS

Scalar data can be passed in memory between overlays that are appended to the main program. The :D command should be used to delete statements that are to be overlaid by new ones.

The :D command and the immediate command "RUN" must be inserted using EDIT.

Example: (PROG1/BAS)

```
10 PRINT "FIRST SEGMENT"
20 GET "PROG2"
```

Example: (PROG2/BAS)

```
10 PRINT "SECOND SEGMENT"
20 PRINT "ENTER 3 NUMBERS: ";
30 INPUT A,B,C
40 APP "OVERLAY"
RUN
```

Example: (OVERLAY/BAS)

```
:D 10-40
10 PRINT "OVERLAY"
20 PRINT A,B,C
30 END
RUN
```

Use EDIT to create file "PROG1/BAS" containing the first three lines of code, file "PROG2/BAS" containing the next five lines of code, and file "OVERLAY/BAS" containing the last five lines of code. Load BASICPLUS and type GET "PROG1". The first segment will be read in and "READY" will be displayed. Type RUN and the first segment will be executed. When line 20 is executed, the next segment will be read in, scratching the segment executing the GET. The RUN at the end of the second segment will cause it to execute as soon as it has been read. Enter three numeric values when requested. They will be passed to the overlay that is appended when statement 40 is executed. When the overlay is loaded, it will execute immediately and display the numbers you entered to demonstrate passage of parameters to appended code.

You can also chain to code generated by your program. For example:

```
10 DIM N$(4)
20 OPEN#2,"MYPROG"
30 PRINT "ENTER N: ";
40 INPUT N$
50 PRINT#2,"10 A=SIN (";N$;")+COS (";N$;")"
60 PRINT#2,"20 PRINT A"
70 PRINT#2,"30 END"
80 PRINT#2,"GO"
90 END#2
100 GET "MYPROG/TXT"
```

When you type RUN, the program will request a number that will be used in the generated program. When you enter the number, the new program will be generated, and automatically loaded and executed.

21.2 Chaining With DOS CHAIN

An extended form of the BASICPLUS command permits the activation of BASICPLUS programs using the DOS CHAIN utility (see the DOS User's Guide). The name of the file containing the program to be loaded into BASICPLUS is specified as a parameter following the command. For example the command:

```
BASICPLS MYPROG;M
```

causes the first program (MYPROG) to be loaded. When BASICPLUS is initialized it will automatically generate a 'GET "MYPROG"' command. Since the command contains a ";M", micro plotting will also be enabled, if a servo printer is on-line.

BASICPLUS <u>will not</u> accept statements from the DOS CHAIN file and must obtain all of its commands with GETs and APPs. Any INPUT(#0) statements will have to be replaced by either INPUT#

<file number> (with the input data stored in a disk file) or by READ and DATA statements written into the program. A GO or RUN statement should be edited onto the end of each BASICPLUS program of the chain to allow its execution without operator intervention. For example:

DOS CHAIN file (CHNFIL):

```
          SNAP PROG1,/CMD
          PROG1
          BASICPLS PROG2
          LIST NEWFILE
```

BASICPLUS Program (PROG2):

```
          10 OPEN#1,"OLDFILE"
          20 OPEN#2,"NEWFILE"
          30 DIM A$(80)
          40 INPUT#1,A$
          50 IF EOF#1 THEN 80
          60 PRINT#2,A$
          70 GOTO 40
          80 END#2
          90 DOS
          RUN
```

Type "CHAIN CHNFIL" to execute the DOS CHAIN file. It will assemble and execute "PROG1". BASICPLUS will be loaded and "PROG2" will be executed. When statement 90 is executed, control will return to DOS CHAIN and the LIST command will be executed. Notice that the input/output form of the END statement must be used. If no number appeared after the END, the program would stop execution and go into Command ("READY") mode awaiting a keyboard command rather than performing the DOS command.

The user can start a program in the Operating System by using

```
    DOS <string expression>
```

For example:

```
    DOS "LIST NEWFILE"
```

will return to the Operating System and the LIST command will be executed.

The user may want to exit BASICPLUS in order to edit his program, using the EDIT command, then return to his program. The following example shows how this can be done.

```
OPEN #1, "CHAINFIL"
PRINT #1, "EDIT BASICPRG/BAS"
PRINT #1, "BASICPLS BASICPRG"
END #1
DOS "CHAIN CHAINFIL"
```

CHAPTER 22. HINTS ON WRITING PACKAGES

BASICPLUS has been designed so that packages that perform useful functions can be written in BASIC. Programs can request that the name of the next program be entered from the keyboard and the string can then be used to GET the next segment or overlay.

If the word RUN or GO is added to the end of the program file with EDIT, the program will be run immediately upon conclusion of loading. If the program is segmented or overlayed, GO should be used to prevent re-initializing everything!

Example of self-starting program:

```
10 PRINT "THIS PROGRAM STARTED ITSELF WHEN"
20 PRINT "IT WAS RETRIEVED FROM THE DISK."
30 END
RUN
```

All user parameters should be checked as closely as possible. Any STOP or END statements should have comments indicating disposition. Be sure that an END is the last statement in the program. Give any directions possible.

Self-destructing programs are also possible. The following is an example:

```
10 PRINT "PROGRAM SELF-DESTRUCTS IN 10 SEC."
20 FOR I=1 TO 500
30 BEEP
40 NEXT I
50 SCRATCH
```

CHAPTER 23. OPTIMIZING USAGE OF WORK SPACE

The program capacity of Datapoint BASIC can be greatly increased by using a few simple space saving techniques when generating programs.

Use FREE to determine the most efficient forms of coding.

Use multiple statements per line when ever possible. (See section 6.3, "Mulitiple-Statement Lines" for restrictions!)

Avoid use of unnecessary REM statements. While REM statements are important for documentation, they do take up space. They should therefore be kept concise.

Keep variable names short, while still intelligible.

Use string characters for storing small positive integers (0-255).

Use GO or GOTO instead of GO TO.

Use "IF <condition> THEN <BASIC statement>" form of IF statement.

Keep message strings in PRINT statements as concise as possible.

Do not use the optional word "LET" in assignment statements unless required.

APPENDIX A. INSTRUCTION SUMMARY

	Page Defined
:D <line number> [- <line number>]	19-2
:M <line number> <old text> <modifier> [<new text>]	19-1
@	18-1
ABS (<numeric expression>)	7-3
AND	9-5
APP <string expr>	20-9
ATN (<numeric expression>)	7-4
AUTO [<line number>] [, <incr>]	20-3
BEEP	11-5
BY	10-1
CAT [# <numeric expr> ,] [<string expr>]	20-6
CLICK	11-5
COS (<numeric expression>)	7-4
CR	18-2
DATA <value> [, <value>]...	13-1
DEF FN <function name> = <numeric expression>	8-1
DELETE # <numeric expression>	17-15
<variable> = DET (<matrix>)	15-10
DIM <variable> <bounds> [, <variable> <bounds>]...	15-1
DOS [<string expr>]	20-5
EF	11-4
EL	11-4
ELSE	9-3
END [" <comment> "]	14-1
END # <numeric expression>	17-6
EOF	13-1
EOF # <numeric expression>	17-7
ERASE [<variable> [, <variable>]...]	20-4
EXP (<numeric expression>)	7-4
FALSE	9-5
FF	18-2
FNn	8-1
FOR <simple num var>=<start> TO <final> [STEP <incr>]	10-1
FREE [# <numeric expr>]	20-6
GET <string expr>	20-8
GO <line number>	9-1
GOSUB [<line number>]	9-2
GOTO <line number>	9-1
HD	11-4

HP (X)	11-3
HU	11-4
IF <cond> THEN <line num> [;ELSE <line num>]	9-3
IF <cond> THEN <stmnt> [;ELSE <stmnt>]	9-3
INPUT [<variable> ,]... <variable>	12-1
INPUT # <numeric expression> , <input list>	17-9
INSERT # <numeric expression> , <string expression>	17-14
INSTR {string}	16-13
INT (<argument>)	7-3
IOPEN # <numeric expression> , <string expression>	17-12
KEY	9-6
LEN (<string expression>)	16-14
LET <variable> = <expression>	7-4
LF	18-2
LIST [# <numeric expr> ,] [<line number>]	20-5
LOG (<numeric expression>)	7-4
MAT <matrix1> = <matrix2>	15-3
MAT <matrix1> = (<scalar>) * <matrix2>	15-5
MAT <matrix> = CON	15-8
MAT <matrix> = IDN	15-9
MAT <matrix> = ZER	15-8
MAT <matrix1> = INV <matrix2>	15-10
MAT <matrix1> = TRN (<matrix2>)	15-9
MAT <matrix1> = <matrix2> <operator> <matrix3>	15-4
MAT INPUT [# <num expr>] <matrix> [, <matrix>]...	15-5
MAT PRINT [# <num expr>] <mat print list>	15-7
MAT READ [# <num expr>] <matrix> [, <matrix>]...	15-6
MAX {arithmetic}	7-1
MAX {string}	16-13
MIN {arithmetic}	7-1
MIN {string}	16-13
NEXT <variable>	10-1
NEXTKEY # <numeric expression>	17-13
NOFILE # <numeric expression>	17-7
NOKEY # <numeric expression>	17-13
NOT	9-5
ON <num expr> GOSUB <line num list> [;<stmt>]	9-7
ON <num expr> GOTO <line num list> [;<stmt>]	9-6
OPEN # <numeric expression> , <string expression>	17-5
OR	9-5
PRINT [<print list>] [<separator>]	11-1
PRINT # <numeric expression> , <print list>	17-8
PRINT USING <format string> , <exp1> [, <exp2>]...	16-7
RANDOMIZE	7-5
RD	11-4
READ <variable> [, <variable>]...	13-1
REM [{any sequence of characters}]	6-2
REN [<line number>] [, <incr>]	20-2

RESTORE	13-1
RESTORE # <numeric expr> [, <numeric expr>]	17-10
RESTORE # <numeric expr> , <string expr>	17-12
RETURN	9-2
ROLLOUT [<line number> ,] <string expr>	20-9
RND	7-3
RU	11-4
RUN [<line number>]	20-1
SAVE [<line number> ,] <string expr>	20-7
SCRATCH	20-4
SCRATCH <string expr>	20-10
SGN (<numeric expression>)	7-3
SIN (<numeric expression>)	7-4
SIZ # <numeric expression>	17-8
SQR (<numeric expression>)	7-3
STEP	10-1
STOP [" <comment> "]	14-1
SUB [<line number>]	9-2
TAB (<numeric expression>)	11-3
TAN (<numeric expression>)	7-4
THEN	9-3
TO	10-1
TO <line number>	9-1
TOPEN # <numeric expession> , <string expression>	17-5
TRUE	9-5
UPDATE # <num expr> [, [<expr> <sep>]... <expr>]	17-14
<format string> USING <exp1> [, <exp2>]...	16-7
VAL (<string expression>)	16-15
VARS [# <numeric expr>]	20-7
VP (X)	11-3

APPENDIX B. RESERVED NAMES

:D	FOR	PRINT
:M	FREE	RANDOMIZE
@	GET	RD
ABS	GO	READ
AND	GOSUB	REM
APP	GOTO	REN
ATN	HD	RESTORE
AUTO	HP	RETURN
BEEP	HU	RND
BY	IDN	RU
CAT	IF	RUN
CLICK	INPUT	SAVE
CON	INSERT	SCRATCH
COS	INT	SGN
CR	INV	SIN
DATA	IOPEN	SIZ
DEF	LEN	SQR
DELETE	LET	STEP
DET	LF	STOP
DIM	LIST	SUB
DOS	LOG	TAB
EF	MAT	TAN
EL	MAX	THEN
ELSE	MIN	TO
END	NEXT	TRN
EOF	NEXTKEY	TRUE
ERASE	NOKEY	UPDATE
EXP	NOT	USING
FALSE	ON	VAL
FF	OPEN	VARS
FNn	OR	VP
		ZER

Reserved names new to version 2 of BASICPLUS are:

INSTR	NOFILE	ROLLOUT
KEY	TOPEN	

APPENDIX C. NUMERIC VALUES OF ASCII CHARACTERS

A	65	a	97	0	48	:	58
B	66	b	98	1	49	;	59
C	67	c	99	2	50	<	60
D	68	d	100	3	51	=	61
E	69	e	101	4	52	>	62
F	70	f	102	5	53	?	63
G	71	g	103	6	54	[	91
H	72	h	104	7	55	\	92
I	73	i	105	8	56	]	93
J	74	j	106	9	57	^	94
K	75	k	107	Blank	32	_	95
L	76	l	108	!	33	@	64
M	77	m	109	"	34	{	123
N	78	n	110	#	35	¦	124
O	79	o	111	$	36	}	125
P	80	p	112	%	37	~	126
Q	81	q	113	&	38	Delete	127
R	82	r	114	'	39		
S	83	s	115	(	40		
T	84	t	116	)	41		
U	85	u	117	*	42		
V	86	v	118	+	43		
W	87	w	119	,	44		
X	88	x	120	-	45		
Y	89	y	121	.	46		
Z	90	z	122	/	47		

APPENDIX D. ERROR MESSAGES

" " UNMATCHED

( ) UNMATCHED

[ ] UNMATCHED

( ) TOO DEEP
The nesting level of parentheses is too deep.

ARGUMENT NOT MATRIX
The argument specified in a MAT statement is not a matrix.

ARGUMENT NOT NUMBER
The argument specified is either a matrix or a string.

ARGUMENT NOT STRING
The argument is not a string.

ARITHMETIC ERROR
The argument of a BASIC function is illegal; i.e., SQR(-4).

BAD FORMAT STRING
The amount of formatted locations and the number of expressions are not the same.

BAD FUNCTION
The function specified in not defined properly.

BAD ISAM FILE
The file used in an IOPEN statement is not properly indexed. INDEX/CMD should be performed.

BAD STATEMENT
A BASIC statement is syntactically incorrect.

CAN'T FIND MATCH
A match for the <old-text> used in a :M statement can not be found.

CAN'T OUTPUT THAT
The user attempted to output something which can not be output by the normal PRINT statement.

DICTIONARY FULL
The dictionary of variable names is filled up. Try SAVEing your workspace and GETting it back to clear memory of variables you no longer need.

DIMENSIONS NOT COMPATIBLE
The dimensions of the matrices are not correct for matrix multiplication, or matrix transposition.

DIMENSIONS NOT SAME
The dimensions of the matrices are not the same for matrix addition, scalar multiplication, subtraction, copying, or inversion.

DIVIDE BY ZERO
Division by zero was attempted.

DRIVE OFFLINE
A reference was made to a drive that is not on line.

FILE DOES NOT EXIST
A reference was made to a non-existant file; i.e., the user tried to SCRATCH a file which did not exist.

FILE IS READ STATUS
An attempt was made to write to a sequential file which has been read from. The user must RESTORE the file, or END the file and re-OPEN it.

FILE IS WRITE STATUS
An attempt was made to read from a sequential file which has been written to. The user must RESTORE the file, or END the file and re-OPEN it.

FILE NOT ISAM
A NEXTKEY, RESTORE by key, UPDATE, INSERT, or DELETE was executed on a file that is not an ISAM file.

FILE NOT OPEN
A reference was made to a file number which had not previously been assigned in an OPEN, TOPEN, or IOPEN statement.

FILE NUMBER IN USE
The user tried to open a file using a file number which is currently assigned to a different file.

FILE SPACE FULL
There is no more room on the disk selected for either another file name or more space for a file.

FOR WITHOUT MATCHING NEXT
A matching NEXT was not found for an exausted FOR loop.

FOR TOO DEEP
A FOR statement has exceeded the maximum nesting level.

FUNCTION TOO DEEP
A user defined function has exceeded the maximum nesting level.

GOSUB TOO DEEP
A GOSUB statement has exceeded the maximum nesting level.

I CAN'T DO THAT
A matrix operation is not in the proper format.

I/O ERROR
The output device has received too many characters for one line, or an illegal device was requested.

INSUFFICIENT CALCULATION SPACE
The matrix inversion or determinant routine could not find enough temporary space to perform its calculations.

INTERRUPTED
The KEYBOARD key was pressed. If a program was executing, the line number of the statement that was interrupted is printed.

INVALID FILE SPEC
The file name used in an OPEN, TOPEN, IOPEN, SCRATCH, SAVE, ROLLOUT, GET, APP, or CAT statement is illegal.

KEY ALREADY EXISTS
The key of an ISAM file used in a INSERT statement already exists.

LINE NUMBER > 38399
The user typed in a line number greater than 38399 or a line number became greater than 38399 during renumbering.

LINE OVERFLOW
The replacement,created a The replacement, insertion, or concatenation in a :M statement, created a new line over 79 characters long.

MATRIX HAS NO INVERSE
The matrix specified in a MAT INV statement has no inverse.

MATRIX NOT SQUARE
The matrix specified in a MAT IDN, DET, or MAT INV statement is not a square matrix.

MISSING OPERATOR
A BASIC statement contains no operator.

TOO COMPLICATED
The user has attempted to have BASIC evaluate an expression which contains too many operators in sequence.

UNDEFINED FUNCTION
The user defined function specified can not be found.

UNDEFINED VARIABLE
The user has attempted to reference a variable which has not yet been assigned a value.

VARIABLE ALREADY DEFINED
The user has attempted to dimension a variable which is already defined as a number.

VARIABLE ALREADY DIMENSIONED
The user has attemented to dimension a variable which has already been dimensioned.

Note: The following error messages occur during input/output operations. The referenced file will be "closed" with no end-of-file mark written. See the DOS User's Guide for explanation of these system errors.

FILE PROTECTION VIOLATION

SYSTEM DATA PARITY FAILURE

RECORD FORMAT ERROR

RECORD NUMBER OUT OF RANGE

NEXT WITHOUT MATCHING FOR
A NEXT statement was executed without a corresponding FOR statement.

NO MORE DATA
The user attempted to read past the end-of-file mark of a file, or tried to READ past the end of DATA statements.

NO ROOM
The user space is full. Try SAVEing your program and then GETting it back. This will clear the work space of variables you no longer need.

NO SUCH LINE
The user attempted to reference a non-existent line number.

OVERFLOW
A number greater than 1E+38 was created.

RETURN WITH NO PRIOR GOSUB
A RETURN statement was executed without a prior GOSUB statement being executed.

SAME MATRIX ON BOTH SIDES
The same matrix was specified on both sides of the equal sign in a matrix multiplication statement.

STACK UNDERFLOW
BASIC tried to execute an illegal statement.

STRING ERROR
The user attempted to read a string into a numeric variable.

SUBSCRIPTING ERROR
The expression used to subscript a matrix or string variable is out of the range of the bounds used to DIM the variable.

The following error messages will occur while loading BASICPLS and will terminate the loading of BASICPLS. For the 5500 version they are:

WRONG PROCESSOR TYPE!
The processor being used does not have a 5500 instruction set.

WRONG DOS!
The D.O.S. is not at least D.O.S. 2.4.

I NEED AT LEAST 36K TO LOAD!
The processor being used does not have 36K of available memory.

For the 1500 version they are:

WRONG DOS!
The D.O.S. is not at least D.O.S. 2.5.

I NEED AT LEAST 48K TO LOAD!
The processor being used does not have 48K of available memory.

If the overlay version of BASICPLS is used, the message:

FAILURE DURING OVERLAY LOAD!

means that the BASICPLS library file is defective and BASICPLS cannot load one of the overlay members. The following error messages occur while rolling-in a BASICPLS object file.

MEMORY SIZE TOO SMALL!
The processor being used has less memory than the processor used to ROLLOUT the program.

OVERLAY LIBRARY MISSING!
BASICPLS is not on the same drive and physical disk location as it was when ROLLOUT was performed.

APPENDIX E. VALID ALPHABETIC CHARACTERS

The following characters are considered alphabetic by BASIC.

A	a
B	b
C	c
D	d
E	e
F	f
G	g
H	h
I	i
J	j
K	k
L	l
M	m
N	n
O	o
P	p
Q	q
R	r
S	s
T	t
U	u
V	v
W	w
X	x
Y	y
Z	z
$	_ (underline)