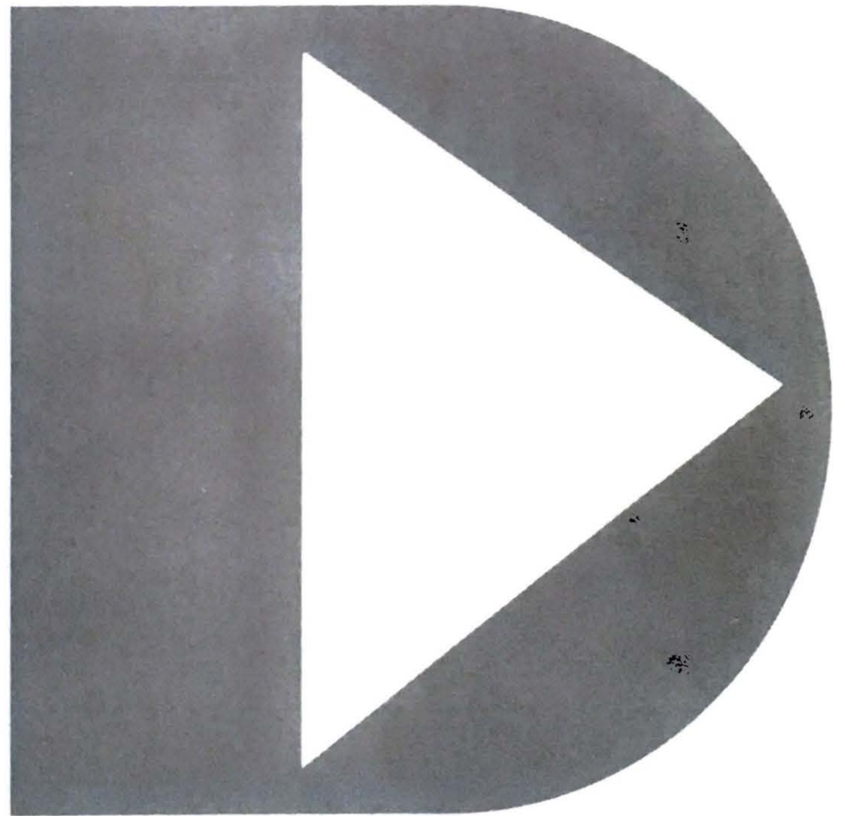
BASIC LANGUAGE
BASIC 2.1
AUGUST 6, 1973

Return to
LOXOP Center

Program Users Guide

Datapoint



COPYRIGHT © 1973 BY DATAPOINT CORPORATION

DATAPOINT BASIC

USERS GUIDE

DATAPOINT CORPORATION

August 6, 1973

TABLE OF CONTENTS

	PAGE
Datapoint BASIC Language Features	1
Manual Conventions	2
Introduction to Datapoint BASIC	3
Modes	11
Constants	12
Variables	13
Implementation Limits on Variables	14
Statement Format	15
Remarks	16
REM	16
Assignment	17
LET	17
Arithmetic Expressions	18
Functions	19
INT	19
LOG	19
EXP	19
SQR	19
RND	19
SIN	19
COS	19
TAN	19
ATN	19
The GO TO Statement	20
The GOSUB and RETURN Statements	21
The IF Statement	22
The FOR and NEXT Statements	24
The INPUT Statement	26
The PRINT Statement	27

	PAGE
The TAB Function	28
The READ, DATA, and RESTORE Statements	29
The STOP and END Statements	30
The RUN Command	31
Utility Commands	32
LIST	32
SCRATCH	32
SAVE	32
GET	32
Advanced BASIC	33
The DIM Statement	33
Use of Arrays	35
Use of Strings	35
Input and Output	38
PRINT #	38
INPUT #	38
LIST #	38
SAVE #	38
GET #	38
BEEP	39
CLICK	39
APP#	39
END #	39
IF EOF #	39
File Hints	40
Formatting Display Output	42
Formatting Servo Printer Output	42
Plotting with Servo Printers	43
Program Chaining	45
Hints on Writing Packages	46
Optimizing Usage of Work Space	47

	PAGE
APPENDIX A (Instruction Summary)	48
APPENDIX B (Numeric Values of ASCII Characters)	49
APPENDIX C (Error Messages)	50

Datapoint BASIC Language Features

<u>Datapoint Corporation BASIC</u> is a fully interactive algebraic language for the Datapoint 2200. While it retains most of the language, simplicity and ease of use of the standard Dartmouth BASIC, it has extensions to take advantage of the unique qualities of the Datapoint. Such extensions provide a complete integration of Datapoint BASIC with all other Datapoint systems and offers the user not only an attractive independent BASIC system, but a powerful additional tool to enhance existing Datapoint systems.

Features include:

* Arithmetic operations (+, -, *, /, ^)
* Arithmetic functions (EXP, LOG, SQR, INT, RND)
* Transcendental functions (SIN, COS, TAN, ATN)
* Logical operators (<, =, >, <=, <>, >=, #)
* Chaining and keyboard controlled execution
* Cassette tape input/output in standard Datapoint file format
* Local, remote or servo printer support configured automatically
* Unused printer support area released and added to user's work area
* Desk calculator execution of expressions
* Built-in debugging aids
* Complete error messages
* Workspace save and restore on cassette tapes in compatible Datatapoint file format
* Long variable names to improve readability
* Multiple statements per line
* String and string array processing
* Co-ordinate micro positioning of servo printer for plotting

Manual Conventions

To help make this manual readable, some conventions are necessary. Corner brackets--< and >--will be placed around words which describe a class of items that should stand in the place of of the corner-bracketed item. For example, the manual will put the legend <digit> in a place where you, the user, can type in any digit.

Square brackets--[and]--will be used to enclose optional 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].

An Introduction to Datapoint BASIC

Since some of the material that follows is rather abstract without some framework, this section will take you through some simple examples. Experienced BASIC programmers may wish to skip this section. Because this section is a hand-holding learn-through-doing text, you will find that it is best read sitting at a 2200.

Place a BASIC tape in the rear deck of the machine and depress the RESTART key on the upper-rightmost part of the 2200 keyboard. This causes the 2200 to load the program that makes BASIC available. It includes the instructions necessary to tell the computer how to calculate all of the arithmetic you may ever use. Besides that are routines that will allow you to take the sine, cosine, tangent, arctangent, natural logarithm, and exponential of any number. The remainder of the tape gives the computer instructions on how to execute the rest of the BASIC language described in the manual.

When the tape is finished reading, it prints out the version of BASIC that you have loaded. Any messages to remind you of release changes are also printed here. 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 variables or programs. Remove the BASIC tape from the rear deck at this time.

The message READY indicates that BASIC is ready to accept a new command. In addition, the cursor, a little block of light that flashes on and off, shows itself at the bottom of 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 depressing the ENTER key (the largest key on the 2200). 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 arithmetic 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
```

to print out the value of the natural logarithm of ten.

The operations and functions include:

```
+ Addition
- Subtraction
* Multiplication
/ Division
^ 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
```

Values can be "remembered" by giving those values to variables. A variable is a named entity whose value can be a number. A variable can be given a value by typing, for instance,

```
LET PI=3.1415927
```

to give the variable named "PI" the value 3.1415927. The variable name PI can now appear anyplace that numbers could appear. Try

```
2*PI
```

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

```
LET PI=-7
```

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

```
PI
```

should print out -7.

Instructions to BASIC can also be laid out as programs. Type

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

and you have stored an instruction that calculates 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 that tells how many times 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 unimportant, but with many statements, the statement number is used to tell the order in which they will be done. 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 can 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 2200 will go through all the stored statements in numerical order. Ten is the only statement, and therefore, it will evaluate the expression you typed in with that number in front of it.

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

```
LET R=.06
```

and getting the new result out with

```
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. And, you have written a one-line program that was stored. 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 we will place a program that computes the time necessary for an object that falls off a desk to hit the floor.

```
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 betwen the double quote marks will be printed. Unfortunately, CALCULATES was misspelled. This, and other errors are corrected by re-typing the line.

```
30 PRINT "THIS PROGRAM CALCULATES THE ELAPSED TIME"
40 PRINT "BETWEEN BEING PUSHED OFF A DESK AND"
50 PRINT "LANDING ON THE FLOOR FOR A HEAVY OBJECT."
```

Note that three separate PRINT statements were necessary to type out the entire message. If you make mistakes, re-type the line in error. You can delete an entire line by typing only the line number.

```
60 PRINT
```

This prints only a blank line.

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

This PRINT statement will type out the material between the double 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 statement: this is the meaning of the semicolon at the end.

```
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/sec2) 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 word SECONDS right next to it. The value of the variable named TIME is printed instead of the letters T,I,M, and E because there are no double quote marks around it. SECONDS, on the other hand, is printed with those letters because it does have double quote marks around it. The semicolon in the middle says to print them 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 have a line number in front of it, it is executed immediately producing a listing of the stored statements that had line numbers in front of them. Note that the program is listed in terms of increasing line numbers even if you had to go back and correct some statements.

It should appear like:

```
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 "BETWEEN BEING PUSHED OFF A DESK AND"
50 PRINT "LANDING ON THE FLOOR FOR A HEAVY OBJECT."
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 printing the answer, BASIC 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 is how long it would take for a penny, for example, to fall off your desk and hit 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. We 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 it to the most compact form possible for storage in the 2200. You cannot, however, put spaces in a variable name like <u>HEIGHT IN INCHES</u>. Fix up the query statement with

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

Try RUNing 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 drop the fractional part left after multiplication -- this will leave an integer and hence the name INT, and (3) print out the word MILLISECONDS after the number of milliseconds. Note that the PRINT statement can print the result of a calculation.

```
LIST
```

Use LIST to list this version of the program. Note that the re-typed lines have been replaced. Try

```
RUN
```

to see how this version works for a penny off your desk. If you wish, you can save this program on a cassette tape for later use. Type

```
SAVE
```

with a casette tape in the front deck and BASIC will save the program there for you. At some later time, you could type GET to retrieve it into a clear workspace.

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 writing down answers without typing RUN over and over again. Try this out by typing

```
RUN
```

and running the program. When you do want to stop the program, the KEYBOARD key on the far right will do the job. Hold it down until the Datapoint says "Interrupted." This key will always get things back to the place where you can type in a command on the keyboard.

The technique just used will type 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? Yes.

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

Typing 80 without any information 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. The NEXT HEIGHTININCHES statement 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 do not know which answer goes with which value of the height. This can be fixed with

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

which will print the height and the time together on one line when executed. The comma between the two values indicates that they should be placed in two columns. The columns can be labeled with

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

which will put labels at the top of the columns. Try this version with

```
RUN
```

Would you like a copy 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 get a copy of the program with LIST #4. If you still have the tape you SAVEd the older version of the program still in the front deck, you can overwrite it, if you wish, by typing SAVE again. Note that the printer must be ON (and ON-LINE if a Local Printer) during initial loading of BASIC since this is when the automatic determination of printer type is made.

Modes

Command mode - This mode is utilized when BASIC has nothing further to do in order to accommodate the last command. A command from the user (such as RUN, LIST, SCRATCH, or a desk calculator command) is required to perform more useful work. This mode is indicated by the word READY and a flashing cursor. Although BASIC does not reply READY, it is also in command mode after accepting a statement to be stored.

Input mode - This mode occurs during an INPUT statement when values for variables are required of the operator. The cursor is flashing. It is the responsibility of the stored program to indicate what information should be entered.

Running mode - During actual execution of instructions, BASIC leaves the cursor turned off and the keyboard inactive.

Depressing the KEYBOARD key at any time will cause BASIC to re-enter command mode regardless of the current mode. Therefore, depressing 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 command referencing the current line (in the case of I/O statements) or the next line (in the case of all computational statements).

Holding down the DISPLAY key at any time will prevent the screen from "rolling up" and losing the top line.

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.

Examples of numeric constants:

```
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.
        Commas are never used in numbers in BASIC.
12E2    has the value 1,200.
        The E is read "times 10 to the"
1.2E6   has the value 1,200,000
1E-2    has the value 0.01.
-2.3E-4    has the value -0.00023.
```

String constants: (Note: String constants are always described to the system in quotes)

```
"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 making them up and the positions they occupy. That is, the constant "YES" can be analyzed by your BASIC program to determine the fact that it is 3 characters in length, that the first is a "Y", that the second is an "E" and that the last is an "S".

Variables

Variables are values that can change. Because of their changeability, they are referred to by names. BASIC variable names begin with a letter. Other letters and numbers can follow if there are no embedded spaces.

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
YEAR-1     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)
```

Variable names cannot begin with REM because those letters are reserved for inserting REMarks in programs. The BASIC operator and command names that cannot be used are:

LET IF FOR THEN NEXT TO GO GOTO INPUT PRINT GET SAVE LIST RUN APP SCRATCH EXP SIN LOG COS SQR TAN ATN GOSUB RETURN DIM AND OR NOT BEEP CLICK TRUE FALSE END STOP INT BY STEP EOF READ RESTORE DATA RND

Hints: Variable names formed with one letter, or one letter and one number are traditional BASIC and cannot possibly conflict with the reserved words listed above. Keeping to the traditional BASIC variable name rule will allow your program to be directly executed on other BASIC systems later, if this is a consideration.

For the advanced programmer: 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). Lower case letters used in variable names are permissible, but the names so formed are distinct from those with different casing. Any combination of lower case letters and numbers forms a valid BASIC identifier because all reserved words are upper-case only. Any symbol beginning with REM is taken as the start of a remark so such variables as REMAINDER should be avoided. Note that the lower case variables are unique to Datapoint BASIC.

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 three bytes (24 bits) of mantissa. All floating point numbers must be 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.

The largest number representable is approximately 1E38. The smallest positive number representable is approximately 1E-38. Precision is ideally 24*(log base 10 of 2) or 7.22 digits. However, values are rounded to 6 digits on output. Note that subtracting similar numbers will lead quickly to a loss of precision. Zero fill is used in normalization.

Overflows during arithmetic will be indicated with the message "OVERFLOW." The characteristic has exceeded the maximum size. Underflow is not announced, and the offending result is placed to zero.

Statement Format

BASIC statements can be stored for later execution, or executed immediately. If a BASIC statement is preceded by a line number and a space, that statement will be stored under that line number. If the statement begins without a line number (0-38399), BASIC will attempt to execute the statement immediately. This latter mode is useful for commands and desk calculator like operations.

Spaces must be used to make BASIC programs readable. For example,

```
LETCOUNTER=7
```

is confusing to you and to the computer. It would be correctly written

```
LET COUNTER=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 operator. The following is ILLEGAL:

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

because spaces occur in the middle of LET and COUNTER.

Multiple statements per line separated by semicolons are permitted. The following are correctly formatted:

```
LET OneMore=OneMore+1
10 LET OneMore = OneMore + 1; A=B=C=0
20 PRINT "HI THERE YOU ALL"
LIST
```

The following are INCORRECTLY formatted:

```
PRINT2+3     space missing after PRINT
104568 HI=1     line number too big
LETA=5     space needed after LET
```

Remarks

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. Although the comments make the program more readable, they take up space that could otherwise be used for active statements.

The REM statement is completely ignored when it is executed. Even the last example will have no effect. Putting valid BASIC 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.

Assignment

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

Examples:

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

An assignment or replacement statement is used to give a value to variables. The value may be a constant or it may be computed as an expression involving any of the arithmetic operations. Parentheses may, of course, be used to form a complicated expression. The word LET is optional and may or may not appear as desired.

Arithmetic Expressions

Arithmetic in BASIC is performed by means of arithmetic operators, arithmetic functions and transcendental functions. They can operate upon:

Numeric constants
Numeric variables

and to be discussed later:

Fully subscripted vectors and arrays
Fully subscripted strings or string vectors

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Raise to a power
INT	Take the largest integer
LOG	Natural logarithm
EXP	Exponential
SIN	Sine
COS	Cosine
TAN	Tangent
ATN	Arctangent
SQR	Square root
RND	Random number

Examples:

<u>2+3*4</u> Operations are done in "normal" mathematical order so that the multiplication is carried out before the addition. "Normal" mathematical order is roughly from the bottom of the list above to the top. If in doubt, use parentheses to force order of evaluation, <u>Result is 14.</u>

<u>(2+3)*4</u> Parentheses forced 2+3 to be evalued first. <u>Result is 20.</u>

<u>(2+5/6</u> <u>Illegal</u> because a right parenthesis is missing.

<u>INT .8</u> <u>Zero.</u> INT finds the largest whole number that is less than or equal to the value of the expression. INT 1 is 1. INT 1.253 is 1. INT -1.9 is -2.

SQR 4 — Two. This can also be written SQR(4).

SIN .1 — Very close to .1. Angle given to SIN and COS is in radians. Likewise, ATN returns a result in radians. Multiply degrees by 3.14159/180 to convert to radians.

4^2 — Four squared. Four is raised to the second power.

SQR(4+12) — Four. The square root of 16.

SQR 4+12 — Fourteen. Without parentheses, the square root is taken first and then twelve is added.

RND 1 — Random result between 0 and 1. The 1 is a dummy argument to satisfy function syntax requirements.

Functions

Form: <function> <expression>

Arithmetic functions:

INT — Gives the largest integer between -8388607 and +8388607 that is less than or equal to value of the expression.

LOG — Gives the natural log of the expression.

EXP — Raises E (E=2.718282) to the power of the expression value.

SQR — Gives the square root of the expression.

RND — Gives a random number between 0 and 1. The expression is a dummy argument.

Trancendental functions: (expression value in radians)

SIN — Gives the sine of the expression.

COS — Gives the cosine of the expression.

TAN — Gives the tangent of the expression.

ATN — Gives the arctangent of the expression.

The GO TO Statement

Form: GO TO <linenumber>
GOTO <linenumber>
GO <linenumber>
TO <linenumber>

Examples:

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

The GO TO statement transfers control of execution. Ordinarily, the BASIC program is executed in line number order. Encountering this statement changes the order of execution. The next statement is the one specified.

The <linenumber> can be specified as a number which represents a line. If, upon execution, no such line exists, the message "No such statement" will be printed and execution will halt. <linenumber> 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.

For the advanced programmer: <linenumber> can be omitted, and the first line of the program is then executed. Therefore, the command "GO" is useful to start programs if the initialization performed by the "RUN" command is not desired. This is commonly the case if extensive work is being done in direct execution mode and indirect mode is entered only as an aid to the direct execution.

When a program has been arrested with the KEYBOARD key, the GO statement can be used to continue execution at a specified line number. The statement printed out at execution arrest is the last statement executed. The GO statement should be to the next line number.

The GOSUB and RETURN Statements

Form: GOSUB [<linenumber>]
RETURN

Examples:

```
GOSUB 19
GOSUB (10*I)
RETURN
```

The GOSUB and RETURN statements can be used to make subroutines in BASIC. The GOSUB behaves exactly like a GO TO except that the next statement number is remembered. When a RETURN statement is executed, the statement after the GOSUB will be executed.

For the advanced programmer: GOSUBs can be nested so that subroutines call subroutines to a limited depth. GOSUB without an argument will GO TO the first statement of the program.

GOSUB can be used in direct execution mode to debug a subroutine. Set up applicable variables. GOSUB <linenumber> in direct execution mode will cause the subroutine to be executed. At the return statement, control is returned to the operator. 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.

The IF Statement

Form:

```
IF <condition> THEN <linenumber>
IF <condition> THEN <BASIC statement>

<condition> is defined as:

     NOT <condition>
     <condition> AND <condition>
     <condition> OR <condition>
     (<condition>)
     <relational>

<relational> is defined as:

     <expression><relational operator><expression>

<relational operator> is:

     <    less than
     <=   less than or equal
     =    equal
     >=   greater than or equal
     >    greater than
     <>   not equal
     #    not equal
```

Examples:

```
IF COST>10000 THEN 50
IF COST>10000 THEN PRINT "Cost too high"
IF A=B THEN GO TO (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 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. Use parentheses to indicate order of complicated conditions.

When the condition is false, BASIC ignores the rest of the statement. When 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 linenumber, BASIC simulates a

GO TO to that linenumber. If the construction does not look to BASIC like a linenumber, what follows THEN is interpreted as a BASIC statement.

Because of the limited precision of computer representation for numbers, computations that come out identical with pencil and paper may not come out identically in the computer. If they were tested for equality, the computer may or may not think that they are equal. Therefore, avoid the use of the equality relational when dealing with fractional quantities. Use >= and <= instead.

For the advanced programmer: Conditions can be tested for truth in direct execution mode. Type only the conditional. BASIC 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. Restriction: the = relational is not available for this use because it conflicts with the use of = in direct assignment statements. (That is, A=B is ambiguous.) Use an equivalent expression like NOT A<>B instead.

The FOR and NEXT Statements

Form:

```
FOR <variable>=<start> TO <final> [STEP <incr>]
FOR <variable>=<start> TO <final> [BY <incr>]
NEXT <variable>
```

Example:

```
FOR ZZZ=1 TO 19
. . . . . .
NEXT ZZZ
```

The FOR and NEXT statements allow repeated execution of the statements between the FOR and the NEXT. <variable> should be a simple variable which will be changed each time the statements contained in the LOOP are executed. It will start at <start>. It will grow by 1 every time through the loop unless a STEP is specified in which case it will change by the amount of the STEP value. As soon as ZZZ goes past the final value, the loop will end and the statement after the NEXT ZZZ statement will be executed.

For the advanced programmer: Loops can run both forward and backward. The default STEP is +1 and the loop runs forward with the <variable> getting larger every time. If STEP is negative, <variable> gets smaller each time. On forward running loops, loop execution continues until the value for <variable> exceeds <final>. When the loop is exited, <variable> contains this value. If the loop runs backward, the loop continues until <variable> becomes smaller than <final>.

Each of the loop specifications may be an expression which change during execution. For example, FOR I=1 TO 100 STEP I will execute the loop 10 times with I taking on the values 1,2,4,8,16,32, and 64. At the exit of the loop, I will have the value 128. The <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 both backward and forward at different times if the sign of the STEP expression is changed and <final> is changed so that the loop will continue.

FOR and NEXT statements 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 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.

Vacuous loops are possible and encouraged.

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.

Samples:

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

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

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

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

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

will print out the numbers from 10 to 1 in reverse order.

The INPUT Statement

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

Examples:

```
INPUT A
INPUT A,B,CCC
INPUT TRIALNUMBER
```

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

One line of input from the operator corresponds to one INPUT statement. If the operator types in fewer numbers than there are variables on the INPUT statement line, the remainder of the variables are given the value zero. If too many numbers are typed in, the excess is discarded.

The format of the input is free. 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.

The PRINT Statement

Form: PRINT [<expression>[<separator><expression>[...]]]

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
PRINT "Name ten Presidents of the United States"
```

The PRINT statement is used to display results of calculations and for informational purposes. In each case, the PRINT statement specifies a list of values to be printed. Each value can be a string or numeric constant or variable. If the word PRINT appears by itself, a blank line is printed.

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

, Comma means "next column"
; Semicolon means "no spacing"

The output page is divided up into columns 16 spaces wide. Separating items with commas tells BASIC to move into the next available column to output the next value. 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 (1) the values will not all fit on one line; then, extra lines will be used or (2) a comma or semicolon terminates the statement in which case the next PRINT statement will continue where the current one stopped.

All printing is done on the CRT of the 2200 in this form of the PRINT statement. Five 16 column groups are spread across the 80 column width of the tube. At most, 12 lines will be visible. Output can be suspended with the DISPLAY key. The screen will not roll as long as it is depressed.

Other forms of the PRINT and INPUT statements are described later.

The TAB Function

Form: TAB <column number>
or TAB (<column number>)

Examples:

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

The TAB function can be used only as a value within a PRINT statement. It causes the next value to be printed in the column number specified to the TAB function. If desired, parentheses can be used around the column number which may be any expression.

TAB can be used to get to particular places on the page for formatting. 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.

<u>No action is taken if the TAB position specified is to the left of the current position of the output.</u>

The READ, DATA, and RESTORE Statements

Form:

```
READ <variable>,<variable>, . . . <variable>
DATA <value>,<value>, . . . <value>
RESTORE
```

Example:

```
READ A,B,C
DATA 10.32,-4,19E5
RESTORE
```

READ behaves much like the INPUT statement except that data is retrieved from DATA statements rather than from the keyboard. <value>s must match the type of the variable used in reading. The RESTORE statement causes the data to be re-read from the beginning. Expressions are not allowed in DATA statements. Unlike INPUT, one DATA statement need not have exactly the number of variables necessary for one READ statement.

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 form calls for the last statement of every program to be an END statment.

The STOP statement is equivalent to depressing the keyboard key. Execution can be resumed with the GO command. END performs the functions of STOP, but in addition, it forces input or output from/to casette tapes to be completed and the tapes used to be rewound.

For the advanced programmer: A common mistake is interrupting a program which is writing a tape and replacing that tape. Any command which will use the new tape will cause BASIC to attempt to complete the old tape, thereby destroying the tape just loaded. The reason for this phenomenon is that BASIC does not know you have replaced the tape. The new command you have issued may very well have applied to the tape currently loaded--in this case you certainly want all of the output written before you start reading it.

You can indicate to BASIC that you are about to replace the current tapes by typing END as a direct command. This will cause any input or output destined for the current tapes to be completed. The new tapes can then be inserted with no fear that deferred input or output will destroy them.

The RUN Command

Form: RUN [<linenumber>]

Examples:

```
RUN
RUN 40
```

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

For the advanced programmer: The initialization steps performed include:

- Write any incomplete output on casette tapes
- Rewind casette tapes before use
- Clear any arrays previously in use
- Clear any memory of previous GOSUBs
- Clear any memory of previous FOR loops
- Cause any READ statements to read from the first DATA statement

Note that simple variables retain their values and are NOT reset to zero or undefined. This permits parameterized execution.

Utility Commands

LIST [<linenumber>] or LIST

The LIST command prints a copy of the current stored statements. If <linenumber> is included, the listing begins at that line. Use the DISPLAY key to hold output on the screen. Use KEYBOARD to cut a LISTing short.

SCRATCH

The SCRATCH command causes everything BASIC knows to be erased. If tapes were being written, they are completed and rewound. The workspace is left clear. The stored program is erased. All variables are erased. This command is useful to get a clean workspace for new work.

SAVE

To save the current stored program, insert a blank tape in the front casette deck and say SAVE. The program will be written to the tape. The values of variables are not written to tape. You will probably want to remove and label the tape immediately.

GET

To recover a stored program saved with the SAVE command, use GET. GET will perform all the functions of SCRATCH except for initializing cassette tapes. The current stored program and the values and names of all variables will be erased. The program in the front deck will be read. If the program was previously SAVEd, the tape will rewind and BASIC will be READY.

Sometimes the 2200 memory can become boggled with variables and values you no longer need. BASIC, however, does not know you no longer need them. You can clean up the memory (so there is more room for new things!) by SAVEing the program and GETing it back.

Complex editing can be performed on the program by SAVEing it and using GEDIT on the resultant tape. The edited program can be retrieved with GET.

Advanced BASIC

The following sections describe in detail features for the advanced BASIC programmer. The user should be thoroughly familiar with the preceding sections before proceeding.

The DIM Statement

Form: DIM <variable>(<bound>) . . . .
DIM <variable>(<bound1>,<bound2>) . . . .

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 that

1. The variable named is an array
2. The maximum size of the array

Collections of numbers can be stored under the same variable name in BASIC. Arrays can be defined as one or two dimensional. In a one dimensional array, numbers are stored and referenced in the same way they would appear in a list. To reference the third number in an array, a subscript of 3 is used. If the array name is A, the reference is A[3]. The square brackets or parentheses are used to enclose the subscript. In a two dimensional array, numbers are stored and referenced the same way they would appear in a table. The row and column which contain the desired number must be specified. The variable A[2,5] references the number in the second row, fifth column of array A.

The <bound> is the number of items that appear in a an array. The first example above saves space for a single dimension array named A which contains 19 items. If two numbers appear between the parentheses, then these are the maximum numbers of rows and columns. In the example above, B is dimensioned as an array with 24 rows and 4 columns.

For the advanced programmer: The value of the <bound>s can be any expression evaluable at the time the DIM is performed. This includes expressions which contain the

results of previous calculations.

A non-dimensioned variable which has previously contained a single, scalar value cannot be redefined as an array and vice-versa. This occurs because BASIC detects an error when the programmer uses the same name for what must be different and independent variables (since they are of different type).

A variable which has been DIMed cannot be DIMed again until a new program is used or a RUN command is given. Some systems do not allow dimensioning of variables by expressions and therefore will ignore DIM statements in execution loops. In Datapoint BASIC, a re-execution of a DIM statement in a program will cause a subscript error. If this occurs, it must then be moved to a position in the program where it will be executed only once. If, in direct execution mode it is necessary to re-DIM a variable, the RUN command with a non-existent 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 2200 memory to allocate space for an array or program. Occasionally, enough space can be recovered by a SAVE and GET to allow execution. Sometimes, 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 go to stored execution, use GO subsequently instead of RUN since RUN deletes DIM storage.

Use of Arrays

A subscripted variable can appear any place where a variable can appear except as the index for a FOR statement. The ordinary parentheses (and) can be used for [and] on input. 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 GO TO (A(2)*7)
70 B(1,1)=5
80 FOR I=C(1) TO B(1,1)
90 C(1+I)=B(I,I)+I
100 NEXT I
110 PRINT A(2),B(1,1),C(6)
```

Use of Strings

The preceding sections have discussed the use of BASIC on numeric data. This section describes the string handling capabilities of Datapoint BASIC. A string of characters can represent names, titles or any other kind of information.

A number that does not change is called a constant. A string that does not change is a string constant. A string constant is written in BASIC by enclosing it in double quote 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).

Strings can also be used as the values for variables. To distinguish between numeric and string variables, the names for string variables end in $. The length of a string must always be given in a DIM statement before the string can be used so that BASIC knows how much room to reserve for it.

```
10 DIM MESSAGE$(80)
20 MESSAGE$="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 while arrays must always be subscripted in use, strings need not be subscripted.

Strings can be used in the following statements:

- LET
- INPUT
- PRINT
- IF .. THEN
- DATA
- READ

The length of a string is the number given to its DIM statement. On input and output, strings of no more than 80 characters can be used. On output, all of the string will be written unless a 13 (new line) or 3 (end of this output) is written into some character position of the string.

The individual characters of a string can be used by subscripting the string just as arrays are subscripted. For example, to print the seventh character of string S$, PRINT S$[7] could be used. Similarly, to replace the 9th character of S$ with a star, use S$[9]="*".

A string is blanked out when DIMed. On a string assignment, excess characters are dropped and extra positions are filled with blanks.

A single character of a string, when subscripted, can act as a real number in calculations. The number will be the value of the character in the ASCII character set. Blank is 32. The letter A is 65. Consult the ASCII chart (Appendix B) for others. Pretend that you want to place the Ith character from A in S$(1). Then, use

```
S$(1)=65+I
```

Examples:

```
10 DIM ANSWER$(3),NAME$(10)
20 PRINT "This program prints backwards."
30 PRINT "Type in the name: ";
40 INPUT NAME$
50 FOR I=10 TO 1 STEP -1
```

```
60 PRINT NAMES$[I];
70 NEXT I
80 PRINT " Do you want to try another? ";
90 INPUT ANSWER$
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$(10),FOUR$(4),NEWFOUR$(10)
30 PRINT "I detect four letter words."
40 PRINT "Try typing in a word"
50 INPUT S$
60 FOUR$=S$
70 NEWFOUR$=FOUR$
80 IF NEWFOUR$=S$ AND (S$[4]<>" ") THEN 110
90 PRINT "That's fine.  Feed me another."
100 GO TO 40
110 PRINT "You typed a four letter word!"
120 FOR I=1 to 100
130 BEEP
140 NEXT I
150 END "I QUIT !!!"
```

Input and Output

Datapoint BASIC allows input and output from several devices. They have been given numbers for use in describing them to BASIC:

0 Keyboard and display
1 Rear casette deck
2 Front casette deck
4 Printer

The input and output statements in BASIC can accept a specifier of the form #<number> to indicate the input/output device desired.

Forms:

```
PRINT #<number>,<expression>. . . . . .
INPUT #<number>,<variable> . . . . . .
LIST #<number>,[<linenumber>]
SAVE #<number>,[<linenumber>]
GET #<number>
```

Note: SAVE and GET #0 uses front casette deck instead of the keyboard/display.

The <number> can be any valid BASIC numeric expression. Input from the printer is, of course, illegal. A handy technique is to use the variable Q to be the output device. Then, during debugging, Q=0 will direct output to the display. During production, Q can be assigned the number of the proper I/O device.

A remote, local or servo printer is automatically configured at system load time. Unused printer support area is automatically added to the user's work area. A remote printer is assumed to be 30 cps through the RS-232 connection of the 2200-400. When using a local or servo printer, make sure it is ON and ON-LINE during load time.

Some extra statements and statement forms for I/O follow:

Form: BEEP
Example: BEEP

The BEEP statement causes the 2200 to beep. This is useful for signaling the operator. Typical uses include signaling errors or the need for operator intervention after a long computation.

Form: CLICK
Example: CLICK

The CLICK statement causes the 2200 to click. This is useful for signaling the operator in situations where the beep would be annoying.

Form: APP [#<number>]
Example: APP

The APP command is used to indicate that the contents of a cassette tape are to be APPended to the current program. If the I/O unit number is omitted or zero, the front casette deck is assumed.

Form: END #<number>
Example: END #2

The END statment with an I/O device number causes the named cassette unit to be re-intialized. This is useful for changing a unit from reading to writing, or vice-versa. For example, assume that an intermediate tape is being written. It should be closed and re-opened for reading. END #2 will close out the front tape and allow it to be read from the beginning without stopping the program.

END #<number> does not stop execution. END without #<number> implies all numbers and in addition, stops execution.

Form: IF EOF #<number> THEN <linenumber>
Example: IF EOF #1 THEN 17

This special form of the IF statement is used to determine when an INPUT statement has encountered an End Of File on a specified casette tape unit. Note: IF EOF # is not a trap and must be executed immediately after the INPUT statement to test the EOF condition.

The following program reads the casette tape in the front deck and prints it on device Q.

```
10 PRINT "List a GEDIT tape on device Q"
20 DIM S$(80)
30 N=0
40 PRINT "PLACE TAPE TO BE LISTED IN FRONT DECK."
50 PRINT "List on Display or Printer? ";
60 Q=-1
70 INPUT S$
80 IF S$="D" THEN Q=0
90 IF S$="P" THEN Q=4
100 IF Q<0 THEN 50
110 INPUT #2,S$
120 IF EOF #2 THEN 200
130 N=N+1
140 PRINT #Q,S$
150 GO TO 110
200 PRINT "- - - - - - -"
210 PRINT "End of file after ";N;" records."
220 END
```

File Hints

The files used by Datapoint BASIC are written so that they are compatible with the CTOS and DOS editors. Therefore, data and programs can be prepared using CTOS GEDIT or DOS EDIT (transfer to and from disk with SIN and SOUT). 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 format 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.

Do NOT read and then write, or write and then read the same casette deck without an END #<number> statement intervening. This can cause some tricky disasters.

Remove cassette tapes when not in use. Punch out the read-only tab on the BASIC cassette. It is too easy to leave tapes in and then accidentally type something that tries to write on the tape.

I/O to tapes not loaded causes the machine to hang. Use the keyboard key to recover. Then type END #<number> for the unit causing problems. This will probably hang again. Repeat the keyboard and END sequence until the machine responds READY. At this point, the machine has finally given up trying to use the empty deck.

Formatting Display Output

The 2200 display can be randomly accessed and maneuvered through BASIC. To do so, special code numbers in strings direct special actions.

08 - A new horizontal position follows
11 - A new vertical position follows
17 - Erase to the end of the screen
18 - Erase to the end of the line

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"
70 GO TO 70
```

Formatting Servo Printer Output

Special code numbers in strings allow paging and over printing on the servo printer:

12 - Page Eject
14 - Carriage return and suppress line feed

For example, the following will overprint one line and page eject:

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

Plotting with Servo Printers

The servo printer can be micro positioned for plotting by using the special co-ordinate positioning feature of 2200 BASIC. The micro co-ordinate feature allows direct positioning to any of 589,824 micro positions on a page before printing. To use this feature reply; "Y" to the message; "WILL YOU BE MICRO PLOTTING?". This will only appear during initialization if your 2200 has a servo printer attached and ready. Any other reply will delete the feature and add the micro plotting area 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
```

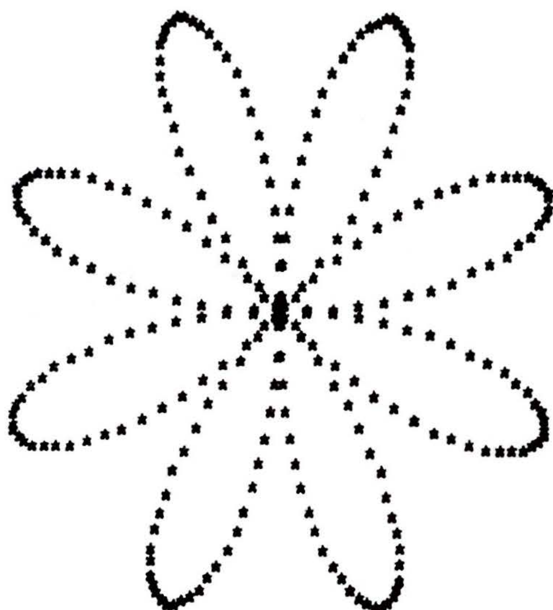
Statement 10 defines a five character string array that will direct the servo printer to position to a micro co-ordinate. Horizontal and vertical co-ordinates must be in the range 0 to +768. A$(1) contains the special function code 15 that indicates to the servo printer driver that co-ordinates follow. A$(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 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
```

RUN the program and you get this:



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!

Program Chaining

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 after one another. If necessary, data can be passed from one program to the other using cassette tape for intermediate storage.

Blank lines and the immediate command "GO" must be inserted using CTOS GEDIT or DOS EDIT (transfer to and from disk with SIN and SOUT).

Example:

```
10 PRINT "FIRST SEGMENT"
20 GET
( blank line )
10 PRINT "SECOND SEGMENT"
20 GET
GO
10 PRINT "THIRD SEGMENT"
20 END
GO
```

Place the cassette containing the edited program in the front deck and type GET. The first segment will be read in, stopping when the blank line is read. READY will be displayed. Type GO (not RUN which rewinds!) 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 GO at the end of the second and third segments will cause immediate execution as soon as they are read in.

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

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

Hints on Writing Packages

BASIC has been designed so that packages that perform useful functions can be written in BASIC. The operator of the function is asked to put the program tape in the front deck and type GET.

If the word RUN or GO is added to the end of the program tape with GEDIT, the program will be run immediately upon conclusion of loading. If the program is segmented, GO should be used to prevent rewinding the program tape!

Example of self-starting program:

```
10 PRINT "THIS PROGRAM STARTED ITSELF WHEN"
20 PRINT "IT WAS RETRIEVED FROM TAPE."
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. 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
```

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 multiple statements per line whenever possible.

Avoid use of REM statements.

Keep variable names short.

Use string arrays for storage of 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 short as possible.

Do not use the optional word "LET" in assignment statements.

APPENDIX A

Instruction Summary

Standard BASIC:	Page
LET <variable>=<expression>	17
GOTO <linenumber>	20
GOSUB [<linenumber>]	21
RETURN	21
IF <condition> THEN <linenumber>	22
FOR <variable>=<start> TO <final> [STEP <incr>]	24
NEXT <variable>	24
STOP ["<comment>"]	30
END ["<comment>"]	30
REM [<any sequence of characters>]	16
DIM <variable>(<bound>)	32
INPUT <variable>,<variable>,. . . .	26
PRINT [<expression>[<separator><expression>[...]]]	27
READ <variable>,<variable>, . . . <variable>	29
DATA <value>,<value>, . . . <variable>	29
RESTORE	29
RUN	31
LIST [<linenumber>]	32
SCRATCH	32
SAVE	32
GET	32

Datapoint Extentions:	
INPUT #<number>,<expression>	38
PRINT #<number>,<expression>	38
BEEP	39
CLICK	39
LIST #<number>,[<linenumber>]	38
SAVE #<number>,[<linenumber>]	38
GET #<number>	38
APP [#<number>]	39
END [#<number>]	39

APPENDIX B

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	
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 C

Error Messages

NO SUCH LINE #
NO ROOM
LINE # > 38399
" " UNMATCHED
DICTIONARY FULL
MISSING OPERATOR
( ) TOO DEEP
( ) UNMATCHED
I/O ERROR
UNDEFINED VARIABLE
STRING ERROR
SUBSCRIPTING ERROR
GOSUB/RETURN ERROR
NO MORE DATA
[ ] UNMACHED
STACK UNDERFLOW
TOO COMPLICATED
BAD STATEMENT
CAN'T OUTPUT THAT
FOR/NEXT ERROR
OVERFLOW
DIVIDE BY ZERO
ARITHMETIC ERROR