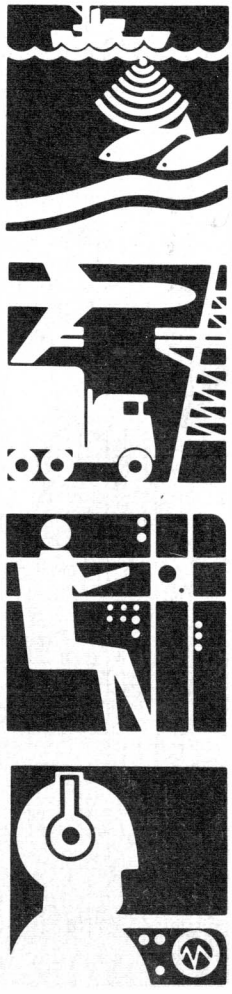


AN INTRODUCTION TO NICOLET BASIC

NIC-80/S-7310-7315

Revised October, 1975

NICOLET INSTRUMENT CORPORATION
5225 Verona Road
Madison, Wisconsin 53711
Telephone: 608/271-3333

TABLE OF CONTENTS

PREFACE

BASIC* is a programming language originally designed for use from a terminal on a large time-sharing system. However, it has proved to be equally useful on all types of systems. As the name implies, BASIC is meant to be simple and easy to learn. BASIC statements are very similar to standard scientific notation and programs tend to be easy to understand simply by reading them. Although the structure and rules of BASIC are relatively simple, it is nonetheless a very powerful programming language, as will be seen in the following sections.

As with most programming languages, BASIC has many versions which differ widely in complexity. The version implemented for the Nicolet 1080 Series is an extended BASIC which is a subset of Dartmouth BASIC. It includes disk file input-output (both sequential and random-access), ability to reference 1080 signal averaging data (a disk unit is required for this feature), and a complete matrix manipulation package.

A description of the features available in Nicolet BASIC is given in Chapters 1 and 2. While some examples of usage are given in Chapter 1, this manual is intended as a reference and user's guide, not as a teaching text. For a more detailed explanation of the various statements, the user is referred to a manual such as "BASIC, 6th Edition," S.V.F. Waite and D.G. Mather, editors, University Press of New England, Hanover, NH 03755.

*BASIC is a registered trademark of Dartmouth College

The following statements are available in Nicolet BASIC:

Operators
+, -, *, /, ↑

Statements
LET
DIM
GO TO
GOSUB ... RETURN
ON ... GO TO
ON ... GOSUB
IF e_1 rel e_2 THEN s
FOR ... NEXT
RANDOMIZE
STOP
END
DEF FNa ($a_1 \ldots a_n$) = e
REM
DATA

Input-Output
FILE #e_1 : e_2
PRINT list
PRINT #e: list (terminal format only)
INPUT list
INPUT #e: list
LINPUT list
LINPUT #e: list
WRITE #e: list (random access only)
READ list (from DATA statement)
READ #e: list (random access only)
MARGIN #e: e_m
SCRATCH #e
RESET #e
IF END #e THEN s
IF MORE #e THEN s

MATRIX Statements
MAT A = B
MAT A = (e) * B
MAT A = B + C
MAT A = B - C
MAT A = B * C
MAT A = CON (e_1,e_2)
MAT A = ZER (e_1,e_2)
MAT A = IDN (e_1,e_2)
MAT A = TRN (B)
MAT A = INV (B)
MAT A$ = NUL$ (e_1,e_2)

Matrix Input-Output
MAT INPUT #e: list
MAT LINPUT #e: list
MAT READ #e: list
MAT PRINT #e: list
MAT WRITE #e: list

Functions
ABS (X) absolute value
ATN (X) arc tangent
COS (X) cosine
COT (X) cotangent
DET determinant of last matrix inverted
EXP (X) e^X
INT (X) integer part of X
LEN (A$) number of characters in string A$
LOC (e) value of file pointer
LOF (e) last character of e
LOG (X) natural logarithm
MAR (e) margin of file e
MEM (I,J) jth word of file I
MOD (I,J) I modulo J
RND random number
SGN (X) +1, Ø, -1 for +, Ø, -
SIN (X) sine
SQR (X) square root
TAB (X) space over to column X
TAN (X) tangent
USER (e_1, e_2) JMS 106630

Directives
NEW
RENAME
LIST
RUN
TAPE
KEY
PUNCH
SAVE
UNSAVE
OLD
DEFINE
DELETE
DIR
BYE

Chapter 1. Introduction to Nicolet BASIC

Loading of BASIC

BASIC is provided in two versions, for disk and non-disk systems. The tapes are identical except that two versions of BASDIR are available -- one marked "disk systems" and one marked "non-disk systems." The non-disk tape removes all references to functions which cannot be accessed without a disk.

The program requires a 12K computer in any case, and loading is accomplished by loading several tapes and storing the result on disk where applicable. The supplied tapes are labeled BASIC1, BASIC2, BASDIR (disk), BASDIR (non-disk) and BASIC.

(a) For non-disk systems

Using the Binary Loader, load the tapes marked BASIC1, BASIC2 and BASDIR (non-disk). The program starts at address zero and can be started by pressing the Stored Program Start button (1080's) or Data Processor Start button (NIC-80's).

(b) For disk systems

Start DEMON/II at 7600, place BASIC1 in the reader and type BIN (Return). When the tape stops, place the BASIC2 tape in the reader and depress Continue and press Execute. When this tape stops, place the BASDIR (disk version) tape in the reader and depress Execute (Continue still depressed).

When this last tape stops, start the computer at 7600 and type

```
STORE BASIC1 0-7577:P
STORE BASIC2 102000-107777:P
STORE BASDIR 100500-101677:P
```

Then place the BASIC tape in the reader and type BIN (Return). When the tape stops, start the computer at 7600 and type

```
STORE BASIC 0-1777;0:P
```

To start the program, type RUN BASIC.

In both cases, the program will start and type out READY. The disk version of BASIC will cause the illegal memory light to flash as it finds out how much memory is available for text storage. In disk systems, the command BYE will cause exit to the DEMON/II monitor.

Getting Started with BASIC

Before we examine a few simple programs, there are some basic concepts which we should familiarize ourselves with. The first of these is the computer version of scientific notation. BASIC will accept numbers both within programs and as values entered into programs in a number of formats. For instance, the number 501 could be entered as

```
      501
      +501
      00501
or    501.00
```

Using the standard computer convention that the symbol "E" means "ten raised to the power," we could also write 501 as

```
      501E0
      50.1E1
      0.501E3
      50100.0E-2
or    5.01E2
```

This last formulation is the computer form of scientific notation and very large numbers or very small ones are always printed out this way by BASIC. However, they can be entered in any format, as long as the total number of significant figures does not exceed eleven.

BASIC has the convention that every line must have an associated line number, and no matter what order the lines are entered in, the program sorts them and executes them in ascending order. Since programs are often likely to be changed by corrections or revisions after they are originally typed up, it has become conventional to begin all BASIC programs at line number 100 and increment the line number by 10 for each successive line. This allows ample space for additional lines where needed.

BASIC, which has the misfortune to be an acronym for Beginner's All-purpose Symbolic Instruction Code, is an algebraic language, like a number of other high level languages. It is similar in concept to the more common FORTRAN, but was specifically designed for time sharing computing and adapts itself well to minicomputer implementation. It is symbolic in the sense that most algebraic statements have a direct analog in BASIC, and thus algebraic manipulations can easily be solved in BASIC.

For example, one can write

```
100 LET Y=M*X + B
```

in BASIC for the more common y = mx + b of algebra. BASIC allows variables as single letter names or single letters followed by numbers. You could also write

```
100 LET Y2 = M4 * X + B7
```

Note the use of the asterisk for the multiplication sign. Similarly, division is represented by the slash, and the equation x = a/b would simply be entered as

```
100 LET X = A/B
```

The other unusual operations symbol available in BASIC is the up-arrow "↑". The usual Teletype and many other teleprinters have the up-arrow character as SHIFT/N. However, a few newer terminals have adopted a new standard and instead have the caret symbol "^" In both cases, this symbol is used to represent exponentiation.

Thus 5^6 is symbolized as 5↑6, (or 5^6). Note that this symbol applies to any base and any power. This is a great deal more general than the E symbol mentioned earlier which is used only to represent 10 to a power. Thus, the value 6.6^3 can only be represented as 6.6↑3, but the value 5×10^3 can be represented as 5E3 or as 5*10↑3. Note however, that the BASIC processor will take much longer to process this second representation and it should be avoided for powers of ten.

Now let us look at some more complex statements. It is always obvious what is meant by a simple statement like y = mx + b. However, a statement like $y = -b + b^2 - 4ac^{1/2}/2a$ may well be ambiguous in meaning. For this reason, in algebra and in BASIC we use parentheses freely to clarify meaning. We write $y = (-b + (b^2-4ac)^{1/2})/2a$ in algebra, or in BASIC:

```
100 LET Y = (-B + SQR(B↑2 - 4*A*C))/2*A
```

Note that the function SQR(X) means the square root of X.

Now, in the case that algebraic parentheses are omitted, BASIC does have a definite order for decoding an expression. This order of priorities is

(1) exponentiation
(2) multiplication or division in the order found
(3) addition or subtraction in the order found

Thus the expression

```
100 LET Y = A + B * C / D - E
```

would be evaluated as if it had been written more clearly as:

```
100 LET Y = A + ( (B*C)/D ) - E.
```

Writing a BASIC Program

Whenever a numerical or logical problem is to be solved, the idea of developing a computer program to solve this problem becomes appealing. However, there are a few rules that one should follow, regardless of the language or simplicity of the program to be written. These are:

1. Define the problem clearly in good English sentences.
2. Draw a flow chart of the method of solution of the problem.
3. Write out the code for the program including copious comments and remarks.
4. Test and debug the program thoroughly.

In most cases, this last step will be the most time consuming. However, the more time that is spent on steps 1 - 3, the less time that step 4 will require. This is not a linear function. Spending x units of time above the minimum on steps 1 - 3 may well reduce step 4 by 2x or 3x.

Let us assume that we are going to write a program to solve the equation $y = mx + b$. The above sentence constitutes step 1. The flow chart is simply the steps:

1. Define values for m, x and b
2. Perform the calculation y = mx+b
3. Print out the result
4. Stop

The code, including comments, would be

```
100 REM SOLVE Y = MX+B.  THIS IS A REMARK
110 LET X = 5
120 LET M = 3
130 LET B = 7
140 LET Y = M*X + B
150 PRINT Y
9999 END
```

In the above program, line 100 is a comment or REMark statement. It has no function in the program but a great deal of value to the programmer. It is printed out when the program is listed and tells the user what the program, or some section of it, is intended to do.

Lines 110 - 130 are simply used to define the values of the constants M, X and B. *BASIC*, like any other language, will make no assumptions. Each value must be clearly known before the equation can be evaluated. Note that each line number is 10 greater than the previous one, leaving ample room for insertions if necessary.

In line 140, the calculation is actually performed. Note that the precedence of multiplication over addition assures that the calculation M*X will be performed before the addition of B. This could only have been circumvented by actually writing Y = M* (X+B), if that was what was meant.

Line 150 introduces the PRINT statement. It simply says print the value of Y on the teleprinter. Line 9999 is an END statement. It is a signal to BASIC to stop execution at this point and that there are no more statements in the program. The last statement of every BASIC program must be an END statement. For this reason, we generally assign the END statement a very high line number, to allow for additional code.

Program Entry and Execution

Now, having prepared the program code, we will enter the program. Upon starting BASIC by either typing RUN BASIC in disk systems or starting at address 0 in non-disk systems, the program will type out READY.

When BASIC types this out, it indicates that it is ready for one of the directives which tell it what to do next. These directives include NEW, LIST and RUN which we will use immediately and a number of others which we will discuss later.

The NEW directive, followed by a program name, enclosed in quotes, tells BASIC that a new program is about to be entered. It clear out any old program text and allows new text to be entered. In this case we will name the program STLINE since this solves the equation for a straight line. Thus, the following events occur.

```
(Start program by typing RUN BASIC or pressing START)
READY
NEW "STLINE"
100 REM SOLVE Y=MX + B.  THIS IS A REMARK
100 LET X = 5
.
.
.
9999 END
```

The program has now been entered. Lines may be typed in any order, but are ordered by the program after entry. If a mistake is made in typing, the RUBOUT key will allow one character to be deleted each time it is struck. It types a backslash for each deleted character until all characters in the line have been deleted. If an entire line is to be retyped the character CTRL/O will remove all text in the current line. If after typing a line, you discover that it is to be changed, typing that line number followed by the new text causes the new text to replace the old. Lines can be deleted completely by typing the line number by itself.

Now, if we wish to obtain a clean listing of the entered program, the LIST command will produce it. After listing it, we can actually execute the program by simply typing RUN. The program will print out

```
RUN
22
STOP
READY
```

The value of mx+b is 22.

Teletype Entry of Values

Now, suppose that we wish to make the above program more useful by entering the value of X at the Teletype each time rather than limiting ourselves to this one value of X. The INPUT statement allows entry of one or more constants at the Teletype. The form of this statement is INPUT X,Y,Z. BASIC will type a ? for each new input statement when the program is run. If several values are required in a single statement, the program will expect them to be separated by commas, but will type only one question mark.

Thus, we can re-write our program as follows:

```
LIST

  "STLINE"
   100 REM SOLVE Y=MX+B
   110 LET M=3
   120 LET B=7
   130 INPUT X
   140 LET Y= M*X+B
   150 PRINT Y
  9999 END
READY
RUN
?7
 28

STOP
READY
RUN
?5
 22

STOP
READY
```

Now if we wish to rerun the program, we simply type RUN again.

Note that we must rerun it from scratch for each value of X. This is clearly tedious and makes little use of the power of BASIC. So, instead of allowing the program to exit each time, we insert the statement GO TO 130 just after 150. The program then loops back and gets a new X after each calculation. Note that this insertion is accomplished by simply selecting the proper statement number, 160.

```
160 GO TO 130
RUN
?3
 16
?4
 19
?6
 25
?7
 28
?9
 34
?Q
ER MXI    130

READY
```

Note that this program has no conditions for stopping and would loop endlessly if not stopped manually by pressing Stop or by entering an illegal number, in this case, Q. A complete program should have exit conditions as well, and we will discuss these later.

The MXI error message, along with all the others, is discussed in Appendix I, page 57.

Use of Loops and Print Statements, GO TO, FOR...NEXT

Let us now suppose that we wish to print out an entire sequence of x,y pairs for the equation y = mx + b. This can be done be printing out values of x and y for each calculated value of the equation. We will also allow the program to ask for M and B at the Teletype. This is easily accomplished using the quoted message facility of the print statement. For example, if we wish to print out M = and then enter the value for M, we need only use the statements

```
110 PRINT "M=";
120 INPUT M
```

The characters enclosed in quotes are printed out when the program is run and the keyboard becomes active, allowing entry of a value for M when the INPUT statement is executed. The semicolon at the end of line 110 means do not start a new line or move over to a new column after the PRINT statement. Normally, each PRINT statement will produce a carriage return and line feed after the required data are printed. This puts the text M = on the same line as the value entered for M.

Now, if we want to print out a whole set of values for Y as X varies from 1 to 10, there are a number of ways to do it. One obvious way would be to set X to 1, calculate Y, print the x,y pair and then increment X. This would be accomplished as follows:

```
140 REM ASSUME M AND B ARE KNOWN AT THIS POINT
150 LET X = 1
160 LET Y = M*X + B
170 PRINT X, Y
180 LET X = X+1
190 GO TO 160
```

Statement 180, LET X = X+1 has a meaning slightly different than in algebra where it would not be allowed. In BASIC, it means "add one to the current value of X and place the result back in X." Thus, the equals sign means "is replaced by" rather than purely "equals."

Note that an x,y pair will be printed out for each value of x, starting at 1 but that there is no upper value of x. In other words, the program will continue to print x,y pairs forever, until $X = 10^{150}$, the upper limit of BASIC. This could take a long time.

Now, since we want to print out only the values of x,y pairs for x between 1 and 10, we must find a way to limit this endless loop. The FOR...NEXT statements in BASIC provide this ability. If we wish to print out all values of Y for X between 1 and 10 we simply write

```
150 FOR X = 1 TO 10
160 LET Y = M*X + B
170 PRINT X, Y
180 NEXT X
```

This sequence of steps will set X equal to 1, perform the calculation, print out the result and, when the NEXT X statement is reached, increment X by 1. The NEXT statement actually means, "advance to the next value of the variable and go back to the top of the loop." A complete program is shown on the following page.

```
LIST

   "XYPAIR"
    100 REM PRINT OUT X,Y PAIRS FOR Y=MX+B
    110 PRINT  "M=";
    120 INPUT M
    130 PRINT "B=";
    140 INPUT B
    150 FOR X= 1 TO 10
    160 LET Y=M*X +B
    170 PRINT X,Y
    180 NEXT X
   9999 END
READY
RUN
M=?7
B=?-4
 1             3
 2             10
 3             17
 4             24
 5             31
 6             38
 7             45
 8             52
 9             59
 10            66

STOP
READY
```

The PRINT statement will automatically produce 5 columns of numbers per line, 15 spaces apart, unless otherwise specified. A more packed output format can be produced by placing a semicolon between items in the print list. Note that retyping line 170 replaces it.

```
170 PRINT X;Y
RUN
M=?7
B=?-4
 1  3
 2  10
 3  17
 4  24
 5  31
 6  38
 7  45
 8  52
 9  59
 10  66

STOP
READY
```

FOR...STEP...NEXT

In the simple case above, the "next" value of X is merely X+1 and the top of the loop is simply the first statement after the FOR statement, or line 160.

The FOR...NEXT statement pair is considerably more versatile than this, of course. If we wished to print out values of x and y for x varying between 0 and 50 in steps of 5, we would write

```
150 FOR X = 0 TO 50 STEP 5
160 LET Y = M*X + B
170 PRINT X,Y
180 NEXT X
```

The starting and terminal values of the FOR loop can be positive or negative and the direction of stepping can also be positive or negative. The value of the STEP increment is assumed to be +1 if it is not specified.

The PRINT statement will also allow arithmetic computations. Statements 160 and 170 could be combined to produce PRINT X, M*X+B as a single statement.

Arrays, DIM, IF...THEN

BASIC is not limited to single variables. It also allows arrays and matrices. These are lists of numbers having one or two indices. The value x_i is represented as X(I) and the value of an element of a matrix a_{ij} as A(I,J), where I and J can be expressions of any complexity.

BASIC automatically reserves 10 locations of storage for an array and 10 x 10 for a matrix, unless otherwise specified. This can be enlarged or reduced using the DIM statement. For example,

```
100 DIM X(5)
```

will reserve 5 locations for X(I) instead of the ten implicit in a reference to X(I) without a DIM statement. This conserves storage in small systems. Similarly DIM X(15) reserves more space.

Let us now consider a program to accept 10 numbers from the Teletype, find the largest one and divide through by it, and print out the resulting normalized list. We will restrict ourselves to 10 numbers, so the DIM statement is unnecessary.

The input section of the program will be as follows:

```
100 REM ALLOW INPUT OF UP TO 10 NUMBERS
110 FOR I = 1 TO 10
120 INPUT X(I)
130 IF X(I) < 0 THEN 200
140 NEXT I
```

This code allows entry of up to 10 positive numbers. If a negative number is entered, the program jumps to statement 200 immediately.

Note the use of the IF...THEN statement at 130. It means if the relation shown is true, then go to the statement number following the THEN statement. If the relationship is not true, go on to the next statement. The operations that can be used here are

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

In the next section of the program we will search for the largest value. This is done by starting at zero and looking for larger values throughout the list.

```
200 REM LOOK FOR LARGEST VALUE
210 LET N = I-1              [MAXIMUM NUMBER OF ENTRIES
220 LET M = 0                [MAXIMUM VALUE
230 FOR I = 1 TO N
240 IF X(I) > M THEN 260     [SKIP IF NOT GREATER
250 LET M = X(I)             [REPLACE IF X(I) > M
260 NEXT I
```

Note that the bracket character (shift-K) can be used in Nicolet BASIC to set off a comment from a BASIC statement. This is useful in calling attention to particular features within a program.

The total number of entries in the array is known from the counter I which will always contain the total number entered plus one. Statement 210 therefore sets N equal to I-1. Thus if 5 numbers were entered, followed by a negative one, X (6) would be the terminating value of the list. Thus the last legal entry would be X(5), or X(I-1).

This program section simply starts setting the variable M equal to zero and then testing each value of the array to see if it is greater than M. If it is, M is set equal to this new large value. After the largest value of M is found, the program goes on to the next section, which will be to divide the total list by this largest

number. This is accomplished by simply stepping through the list and dividing each value by M. After each value is normalized, the program prints it out. This is shown below.

```
 300 REM DIVIDE EACH ENTRY BY M
 310 FOR I = 1 TO N
 320 LET X(I) = X(I)/M
 330 PRINT X(I)
 350 NEXT I
9999 END
```

Obviously, the same techniques are applicable to academic grade-keeping and other bookkeeping tasks. The entire program along with its comments and output is given below. Note that BASIC represents the small number, .0625, in scientific notation as 6.25000E-2.

```
LIST

  "NORML7"
   100 REM ALLOW INPUT OF UP TO 10 NUMBERS
   110 FOR I = 1 TO 10
   120 INPUT X(I)
   130 IF X(I) < 0 THEN 200          !EXIT IF NEGATIVE
   140 NEXT I
   200 REM LOOK FOR LARGEST VALUE
   210 LET N= I-1                   !MAXIMUM NUMBER OF ENTRIES
   220 LET M=0                       !MAXIMUM VALUE
   230 FOR I = 1 TO N
   240 IF X(I) < M THEN 260          !SKIP IF NOT GREATER
   250 LET M = X(I)                  !REPLACE IF X(I)>M
   260 NEXT I
   300 REM DIVIDE EACH ENTRY BY M
   310 FOR I = 1 TO N
   320 LET X(I)= X(I)/M
   330 PRINT X(I)
   340 NEXT I
  9999 END
READY
RUN
?5
?3
?7
?1
?16
?8
?9
?2
?-1
 0.312500
 0.187500
 0.437500
 6.250000E-2
 1
 0.500000
 0.562500
 0.125000

STOP
READY
```

Use of Strings in BASIC

While BASIC has the ability to handle single numbers and arrays as many other languages do, it also has an exceedingly powerful ability to manipulate characters in groups called strings. We have already seen one example of a string as a unit of text between quotes in the PRINT statement. In addition, text characters can be entered at the Teletype and manipulated much as variables can.

In BASIC, a string variable is given a one- or two-character name followed by a dollar sign. Strings may be up to 15 characters in length. Some large computer versions of BASIC allow much longer strings than 15 characters, but this limitation was imposed in Nicolet BASIC to minimize wasted storage.

Now, suppose we wish to allow the user of a program to answer YES or NO to a particular question. We handle this by asking for the input of a string variable and testing for its equality with "YES" and "NO." If the input variable is equal to neither, the question should be repeated. We will consider only the fragment of the program which performs this function.

```
200 PRINT "ANS YES OR NO ";     [ABBREV. LIMITS LENGTH TO 15 CHARS
210 INPUT A$                    [A$ IS A STRING VARIABLE
220 IF A$ = "YES" THEN 300
230 IF A$ = "NO" THEN 400       [BRANCH TO 300 OR 400 IF YES OR NO
240 GO TO 200                   [IF NEITHER, PRINT QUESTION AGAIN

300 ...
400 ...
```

Note that the string variable A$ is compared to a character string within quotes. It could also be compared to another stored string as follows:

```
100 LET N$ = "NO"
110 LET Y$ = "YES"

200 PRINT "ANSWER ";"YES OR NO ";   [TWO STRINGS ATTACHED WITH ";"
210 INPUT A$
220 IF A$ = Y$ THEN 300
230 IF A$ = N$ THEN 400
240 GO TO 200
```

This second formulation would enable comparison to the variable N$ and Y$ at several places within the program without creating new strings and wasting storage. Note the limitation to 15 characters necessitates abbreviating the string in the first example to "ANS. YES OR NO". Note also that the answer will be on the same line as the question, as the carriage return is suppressed at the end of the print statement using the semicolon.

In the second example, two short strings are printed next to each other without spaces by placing a semi-colon between them. This avoids having to abbreviate as in the first example.

A more useful example of the capability of handling character strings and numbers interchangably is shown in the example below. The program converts Fahrenheit to Centigrade and vice versa. If the number entered is followed by a C, the temperature is assumed to be Celsius and converted to Fahrenheit. If the number entered is followed by an F, it is converted from Fahrenheit to Centigrade. If a Q is typed, the program halts.

```
LIST

  "TEMP  "
   100 REM FAHRENHEIT TO CENTIGRADE CONVERSION
   110 INPUT T, A$          [GET TEMP AND TYPE
   120 IF A$ = "F" THEN 200
   130 IF A$ = "C" THEN 300
   140 IF A$ = "Q" THEN 9999    [Q MEANS QUIT
   150 PRINT "PLEASE TYPE ";"ONLY F OR C"
   160 GO TO 110
   200 LET T= (5/9) * (T-32)
   210 PRINT "=";T;"C"
   220 GO TO 110
   300 LET T= 1.8 * T +32
   310 PRINT "=" ; T ; "F"
   320 GO TO 110
  9999 END
READY
RUN
?85C
= 185 F
?-40C
=-40 F
?12L
PLEASE TYPE ONLY F OR C
?-24F
=-31.1111 C
?10Q

STOP
READY
```

Note that the INPUT statement at 110 allows entry of both the temperature and the string A$. If A$ is equal to anything other than F, Q, or C, the program types PLEASE TYPE ONLY F OR C. This is the sort of drop-through filtering common in computer programming. The character is tested for all legal values and sent to various program points if a legal character is found. If one is not found, the program reaches statement 140, prints an error message and returns to 110 for a new entry.

String Arrays, the RND Function

One possible use for string arrays is printing out messages which vary from time to time and the index of a particular element in the string will be determined within the program. One exceedingly useful application of this feature is in "fuzz-phrase" generators, so common in bureaucratic and technical language.

We will assume that a table of adverbs is stored in A$(I), a table of adjectives in B$(I) and a table of nouns in C$(I), where each of these is 10 entries in length. Then we need only generate a random index between 1 and 10 which will enable us to reach one entry in each list in a random fashion. The result will be, for instance A$(5) followed by B$(7) followed by C$(2). We will cycle through this list 50 times to produce 50 good fuzz phrases.

Now, if we wish to produce a random number in a specified range, the RND function can be used. RND produces a number between 0 and 1 each time it is called and we need only scale this to the proper range. This can be done by the equation

```
200 LET I = INT(RND*N+1)
```

Using this equation, if RND returns 0, I will be 1 and if RND returns .999, I will be 10. The function INT means the integer part of the expression shown. Note that the function RND has no arguments.

Now there are three such numbers needed, each one an index for one of the tables. This means that the calculation shown in statement 200 above must be performed for three values of RND. This can be accomplished in several ways.

One way would be to simply write:

```
.
200 LET I = INT(N*RND+1)
210 PRINT A$(I),
220 LET I = INT(N*RND+1)
230 PRINT B$(I),
240 LET I = INT(N*RND+1)
250 PRINT C$(I)
.
.
.
```

Subroutines

This is clearly inefficient and wasteful of typing time if not of computer storage space. Instead, let us consider the possibility of writing a subroutine to calculate this parameter.

In BASIC, a subroutine is simply a statement or group of statements which can be executed from several places in the main program by a GOSUB command. The entry point of a subroutine is simply the first statement to be executed. It has no other special characteristics. The subroutine will return to the statement following the call by simply giving the RETURN statement as shown below.

```
200 GOSUB 300
210 PRINT A$(I),
220 GOSUB 300
230 PRINT B$(I),
240 GOSUB 300
250 PRINT C$(I)
.
.
300 LET I = INT(N*RND+1)
310 RETURN
```

Now, the actual calculation of I is performed at only one point, but used to print out random entries in the A$, B$ and C$ tables. Note the commas terminating lines 210 and 220 which suppress the carriage return at the end of the print statement, but cause spacing to the beginning of the next 15-character column. The complete program for generating chemical fuzz phrases is shown below, with output following.

```
LIST

  "FUZZ  "
   100 REM FUZZ PHRASE GENERATOR
   103 PRINT "N=";
   105 INPUT N
   110 FOR I= 1 TO N
   120 GOSUB 180              [GET EACH ADV, ADJ AND NOUN
   130 NEXT I
   140 PRINT "CHANGE:";
   150 INPUT I              [GET LINE NUMBER TO CHANGE
   160 IF I<= 0 THEN 200
   170 IF I>N THEN 200       [GO ON TO PRINT IF OUT OF RANGE
   172 GOSUB 180
   174 GO TO 140
   177 REM INPUT SUBROUTINE
   180 PRINT I;
   185 INPUT A$(I),B$(I),C$(I)
   190 RETURN
   200 REM PRINT OUT 50 FUZZ PHRASES
   210 FOR J=1 TO 50          [J IS COUNTER OF RANDOM PHRASES
   220 GOSUB 300
   230 PRINT A$(I),
   240 GOSUB 300
   250 PRINT B$(I),
   260 GOSUB 300
   270 PRINT C$(I)
   280 NEXT J
   290 STOP
   300 REM RANDOM NUMBER SUBROUTINE
   310 LET I=INT(RND*N +1)
   320 RETURN
  9999 END
READY
```

```
RUN
N=?10
  1 ?HIGH ENERGY,HYPER,LOCALIZATION
  2 ?MULTIVALUED,SPECTRO,INVERSION
  3 ?ANTIPARALLEL,SUPER,CONJUGATION
  4 ?BIPLANAR,POLY,COUPLING
  5 ?THEORETICAL,THERMO,DETECTION
  6 ?ROTARY,ANTI-,ROTATION
  7 ?SOLVATED,ELECTRO,INTERCHANGE
  8 ?DEPROTONATED,INFRA,VISCOSITY
  9 ?OCTAHEDRAL,CHEMO,LUMINESCENCE
  10 ?OPTICAL,META,DISPERSION
CHANGE:?-1
THEORETICAL     SPECTRO         LUMINESCENCE
OPTICAL         POLY            INVERSION
ANTIPARALLEL    HYPER           LUMINESCENCE
THEORETICAL     POLY            DETECTION
ANTIPARALLEL    HYPER           DISPERSION
BIPLANAR        SPECTRO         VISCOSITY
MULTIVALUED     SPECTRO         INTERCHANGE
MULTIVALUED     THERMO          VISCOSITY
OPTICAL         META            DETECTION
SOLVATED        ANTI-           LOCALIZATION
ANTIPARALLEL    META            INTERCHANGE
ROTARY          CHEMO           LUMINESCENCE
THEORETICAL     THERMO          INTERCHANGE
SOLVATED        INFRA           DISPERSION
DEPROTONATED    THERMO          DISPERSION
DEPROTONATED    ELECTRO         LUMINESCENCE
HIGH ENERGY     INFRA           INTERCHANGE
OPTICAL         INFRA           INTERCHANGE
BIPLANAR        HYPER           VISCOSITY
OCTAHEDRAL      SUPER           VISCOSITY
MULTIVALUED     HYPER           DISPERSION
OCTAHEDRAL      ELECTRO         VISCOSITY
OPTICAL         THERMO          DISPERSION
ANTIPARALLEL    ELECTRO         DISPERSION
BIPLANAR        POLY            LUMINESCENCE
MULTIVALUED     ELECTRO         INTERCHANGE
BIPLANAR        ELECTRO         VISCOSITY
OCTAHEDRAL      SUPER           DETECTION
MULTIVALUED     ANTI-           INTERCHANGE
HIGH ENERGY     META            INTERCHANGE
HIGH ENERGY     SPECTRO         INTERCHANGE
OCTAHEDRAL      POLY            INTERCHANGE
HIGH ENERGY     CHEMO           DETECTION
ROTARY          THERMO          INVERSION
ANTIPARALLEL    HYPER           CONJUGATION
ANTIPARALLEL    SUPER           LOCALIZATION
OCTAHEDRAL      HYPER           DETECTION
HIGH ENERGY     SUPER           INTERCHANGE
OPTICAL         POLY            INTERCHANGE
OPTICAL         HYPER           LUMINESCENCE
ANTIPARALLEL    POLY            VISCOSITY
OCTAHEDRAL      SUPER           ROTATION
HIGH ENERGY     CHEMO           CONJUGATION
DEPROTONATED    HYPER           VISCOSITY
ROTARY          CHEMO           INVERSION
BIPLANAR        SPECTRO         LUMINESCENCE
ANTIPARALLEL    INFRA           LOCALIZATION
BIPLANAR        ELECTRO         LUMINESCENCE
OCTAHEDRAL      INFRA           INVERSION
ANTIPARALLEL    SPECTRO         CONJUGATION

STOP
READY
```

The above program "FUZZ" has three sections, an input section, a modification section and an output section. In the input section the number of entries, N, is determined by printout N= at the Teletype and allowing entry of a number. Note in the program output that it actually prints N=?. This occurs because BASIC always prints a question mark to indicate that it is waiting for data to be typed in.

The program cycles through statements 110-130 N times, acquiring N values for A$(I), B$(I), and C$(I). The actual INPUT statement is located at 180-190 where the current value of I is printed out and values are accepted for A$(I), B$(I) and C$(I). Note that the semicolon following the PRINT I; statement prevents a carriage return or spacing over to a new column.

After the data are all entered, the program provides an opportunity to correct typographical errors by printing CHANGE: and asking for a value of I. It then tests I for being in range. If I is less than 1 or greater than N, the program proceeds to the output section. If I is within range, the program calls the input subroutine again and allows entry of a new line of three words.

In the output section, starting at 200 the program uses J as a counter of the number of fuzz phrases. J does not appear as an index of an array, but simply counts lines. The random number subroutine is called, which returns a value of I between 1 and 10. Then the A$(I) word is printed; some word between A$(1) and A$(10) depending on the value of I. A new value of I is fetched twice more to determine a random member of the B$ and C$ arrays and each of the three words is printed. They are all printed on the same line since the PRINT statements end in a comma. After 50 lines are printed the program stops.

It should be noted that it is perfectly legal to reference a REMark statement by number. Statements 160 and 170 both test the range of I and if not in range send the program to statement 200, a REMark. However, since remark statements are not executed the program executes statement 210 next. The branching to a remark is common in BASIC as the destination statement then can be used to define the function of that branch of the program.

General Input/Output Discussion

The PRINT statement, if used to print out a list of data, as in PRINT A,B,C, D,E$, prints one number or character string for each 15 column field. This is somewhat wasteful of space and of printer time if small integers or strings are to be printed. To print numbers as close as possible together, the items in the list can be separated with semicolons as in PRINT A;B;C;D;E$. This will cause only one space to be printed between the end of one item and the beginning of the next.

The end of each PRINT statement automatically produces a carriage return and a line feed. A PRINT statement with no list following will produce a carriage return-line feed by itself, or in other words, a blank line. The carriage return-

line feed can be suppressed by following the last element in the list with a comma or a semicolon. If a comma is used, the printer will space over to the start of the next 15 character column when more printing is requested on the same line. If a semicolon is used, only one space will be produced between data printed in separate statements.

The INPUT statement will print one question mark for the first item in the list: INPUT A,B,C$,D,E and will expect to find a comma separating other elements entered on the same line.

Data can be entered each on a separate line using the LINPUT statement which will print a question mark for each item and expect to find it on a new line. Further, the LINPUT statement can be used to read in character strings which contain commas.

Thus, the statement

```
100 LINPUT A,B,C$,D,E
```

expects to find numbers A and B on separate lines, string C$ on a third line and numbers D and E on fourth and fifth lines. The string C$ could contain a comma if the user wished.

Multiple Branching ON...GO to and ON...GOSUB

We have seen how to use the IF...THEN statement to branch to different program points depending on whether the IF condition is fulfilled or not. It is also possible to branch to several points depending on calculated conditions, using the ON...GO TO statement. This statement works as follows.

```
100 ON expression GO TO 200,300,400
```

The expression is evaluated and if its integer part is equal to 1, control is transferred to the first statement number listed, if equal to 2 to the second and so forth. A number calculated for the expression which is out of range produces an error message and program interruption.

Now let us consider a simple example of the use of this expression. The SGN(X) function returns -1 if the number X was negative, 0 if it was 0 and +1 if positive. The statement

```
100 ON SGN(X)+2 200,300,400
```

jumps to 200 if X is negative, 300 if X is 0 and to 400 if X > 0.

Other uses of the ON...GO TO statement might include the ability to perform one of a number of arithmetic operations depending on a value typed at the teleprinter.

For instance, if a 1 is typed, do one thing, if a 2 is typed, do another, and so forth.

Exactly analogous to the ON...GO TO statement is the ON...GOSUB statement which jumps to one of a number of subroutines depending on the calculated value of the expression.

Formatting Output

The PRINT statement prints data in 15 character columns when the elements of the list are separated by commas and one space apart when separated by semicolons. However, this rather primitive formatting arrangement does lead to some uneven columns. Instead, we have the option of using the TAB function to move the printer head to a particular column. For instance, PRINT TAB(5) would print five spaces if the printer head were at the left edge and no spaces if already at position 5. The number in the parentheses may be either an integer or an expression, but is always an absolute position. It means move to the n^{th} column, not move over n more columns. The simple program on the next page will print a tree and branch out into bad puns.

Matrix Statements

BASIC contains rather complete matrix manipulation facilities allowing complex transfers to be taken care of in single statements. All matrix statements are in the form

```
100 MAT operation
```

where the matrix operations include addition, subtraction, multiplication, inversion, constant transfer and printing.

For a matrix operation to be performed, the matrices used must have been previously dimensioned, using a DIM statement. The arithmetic operations available are

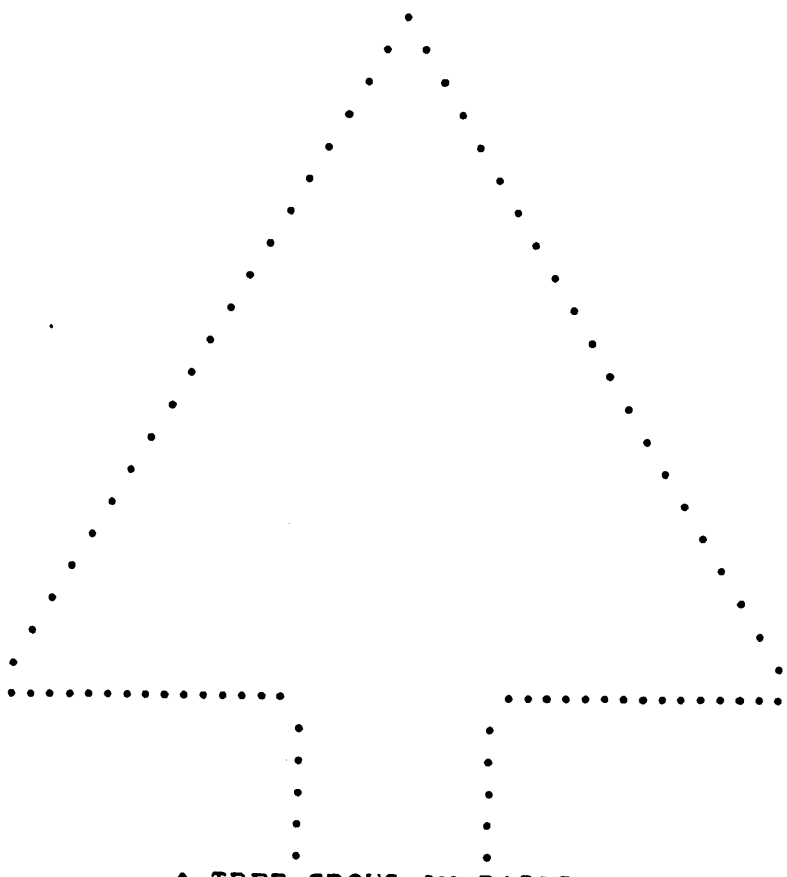
transfer	100 MAT A = B	
addition	110 MAT A = B+C	
subtraction	120 MAT A = B-C	
constant multiplication	130 MAT A = K*B	where K is a constant
matrix multiplication	140 MAT A = B*C	

In the above, A, B and C are assumed to be all matrices or all vectors (one dimensional arrays). The same matrix can never appear on both sides of an equation.

```
OLD "TREE"

LIST

  "TREE  "
   100 REM "OH, TANNENBAUM!"
   110 LET A$="."
   120 PRINT TAB(30);A$
   130 FOR I = 1 TO 20
   140 PRINT TAB (30-I);A$;TAB(30+I);A$
   150 NEXT I
   155 PRINT TAB(10);
   160 GOSUB 300
   170 PRINT TAB (36);
   180 GOSUB 300
   185 PRINT          [CARRIAGE RETURN
   190 FOR I = 1 TO 5
   200 PRINT TAB(25); A$; TAB(35); A$
   210 NEXT I
   220 PRINT TAB(19);"A TREE GROWS ";"IN BASIC"
   230 STOP
   300 FOR I = 1 TO 15
   310 PRINT A$;
   320 NEXT I
   330 RETURN
  9999 END
READY
RUN
```



```
A TREE GROWS IN BASIC

STOP
READY
```

BASIC also allows several matrix functions, which are adequately described in the specifications chapter. They are:

set to +1	100 MAT A = CON
set to 0	110 MAT A = ZER
identity	120 MAT A = IDN
transpose	130 MAT A = TRN(B)
inverse	140 MAT A = INV(B)

BASIC also allows entire matrices to be entered or printed out in a single statement using MAT output commands. These are:

```
110 MAT INPUT A
120 MAT LINPUT A
130 MAT READ A
140 MAT PRINT A
150 MAT WRITE A
```

Note that an entire array can be listed out in this way, using a MAT PRINT statement, rather than a cumbersome FOR...NEXT loop. An example is shown below:

```
LIST

  "MATFUN"
   100 DIM X(7)
   110 MAT INPUT X
   120 MAT PRINT X;
  9999 END
READY
RUN
?1,2,3,4,5,6,7
 1  2  3  4  5  6  7

STOP
READY
```

All of the above input/output statements can be used for disk file manipulation, as shown in the next chapter.

DEF and FNa - Defining Functions

Occasionally a calculation too short for a subroutine must be performed several times in a BASIC program. For instance, the function which converts a number into 1, 2 or 3 depending on its sign, can be defined and used throughout a program. The function definition can appear anywhere in the program.

This definition has the form

```
100 DEF FNa (x,y,..) = expression
```

The first two letters of a defined function name are always FN, and the third is selectable by the user. The number of arguments within the parentheses depend on the number of variables in the expression, and are considered "dummies." That is, the actual variables need not be x and y, but variables must be entered there which will be used in that order in evaluating the function. For instance, in the following program, we will combine an FNa and an ON...GO TO statement to test a number typed at the Teletype for being negative, positive, or zero.

```
100 DEF FNA(X) = SGN(X) + 2        [FUNCTION DEFINITION
110 INPUT A
120 PRINT "THAT NUMBER IS ";
130 ON FNA(A) GO TO 140, 160, 180
140 PRINT "NEGATIVE"
150 GO TO 110
160 PRINT "ZERO"
170 GO TO 110
180 PRINT "POSITIVE"
190 GO TO 110
9999 END
```

Note that X is used as a dummy argument in the DEF statement, but that the actual function operation is performed on the value of A. In fact, the value of X is never defined anywhere, and need not be unless it appears in an executable statement.

Saving and Reusing Programs - SAVE, NEW, PUNCH, OLD

Obviously the labor of writing a BASIC program is only worthwhile if it can be saved and reused whenever a set of data is to be analyzed. There are several BASIC directives which can facilitate program reuse. Recall that a directive is some instruction given the BASIC processor following a READY. These so far have only included NEW, RUN and LIST.

Another directive which can be used in disk systems is SAVE. The SAVE directive will write a copy of your program onto disk using the name you used after NEW and give it the .B suffix. For example, the STLINE program which was created after typing

```
NEW "STLINE"
.
.
PROGRAM TEXT
.
SAVE
```

would be saved as STLINE.B in the DEMON directory. The B extension means, of course, that it is a BASIC file.

In non-disk systems, the program can be punched out on the low speed punch by simply typing LIST, and turning on the punch before typing a Return. If a high speed punch is available, the command PUNCH will punch the tape out on the high speed punch.

Conversely, any file that has been SAVEd can be deleted from disk while within BASIC by typing the rather ungrammatical UNSAVE, followed by a file name in quotes.

A tape which was punched out using the PUNCH command can be read in at the high speed reader using the TAPE directive. The KEY command, read in at the high speed reader, means return to the Teletype. Consequently, the last line on any such tape must be the command KEY. This is automatically produced by the PUNCH command, but tapes generated in other fashions, such as by turning the punch on during LIST, or preparing a tape off line, must have this command manually appended.

Once a program such as STLINE has been SAVEd on disk, it can be recalled and run by simply giving the directives

```
OLD "STLINE"
RUN
```

This will call the program into memory for execution, and run it.

Chapter 2. Use of the Disk in BASIC

The BASIC Directory

Disk BASIC has the capability of accessing data stored on disk whether produced by previous BASIC programs or stored by DEMON. It can access any file stored on disk, assuming that it has a .B extension. However, to minimize disk look-up time, BASIC maintains its own internal directory of 32 files which can be accessed by any one program. These are entered in the BASIC directory, BASDIR, automatically by BASIC using two commands.

The command DEFINE followed by a filename in quotes causes BASIC to look up a file in the DEMON directory and write it into the BASIC directory. This directory can then be printed out using the BASIC command DIR.

For example

```
DEFINE "BFILE"
```

Thus, the DEMON file named BFILE.B is added to the BASIC directory, where the entries are name, size in characters, type, contents in characters, and margin, the column width. The BASIC file types are U for undefined, TER for terminal format, NUM for random access numeric and STR for random access string. A file is always empty and undefined when DEFINEd by BASIC. The file BFILE can then be accessed and used within the BASIC program as described below.

BASIC Disk Files

BASIC files on disk are typed as terminal-format, random access numeric or random access string. When a file is created without being written into or when it is zeroed, the type becomes undefined. It is defined by the nature of the first information written into it.

A random access numeric file consists of a linear array of numbers which can be accessed by index number. Thus, we could write A(I) for I equal to 1 through 2000 into a random access numeric file. This would reduce the amount of internal storage necessary for long arrays, but still make them available to BASIC from disk. By random access, we mean that we can obtain or write any single element or group of elements in that list of A(1) through A(2000). Such a file, then, is treated as merely a list of numbers which can be read and written as needed onto disk.

Similarly, a random access string file is a list of string expressions which can be indexed and accessed as needed. Such an index might include input to

the fuzz phrase generator we wrote earlier or several types of input depending on the field of endeavor to be spoofed. Different files could then be used as desired.

In order to use a file named ABCDEF in a BASIC program, there are three levels of definition necessary.

1. There must be a file in the DEMON directory named ABCDEF.B.

2. The file ABCDEF must be DEFINEd using the BASIC directive
   DEFINE "ABCDEF"
   to enter it in the BASIC directory.

3. The file must be given a number within the program, using the FILE statement
   FILE #1:"ABCDEF"

The above three rules are true regardless of whether the file is to be read from or written into. This means you cannot write data from a BASIC program into a file unless it already exists in the DEMON directory. To create a file, any data can be used, such as the position of the buttons. A 4K file will allow 2048 numbers or 800 strings, and larger sizes proportionately more. Remember that such a file must be stored including the .B extension.

The FILE Statement

While both DEMON and the BASIC directory refer to files by six character names BASIC programs refer to files by number. This convention was adopted because a file number can then be the result of evaluating an expression as well as an integer referenced directly. Thus, file numbers can be variables which change depending on conditions within the program if the user wishes.

The BASIC statement

```
100 FILE #1: "ABCDEF"
```

will define file number 1 as that one named ABCDEF in the BASIC directory and ABCDEF.B in the DEMON directory. Once the file number has been defined, the READ# and WRITE# statements can be used to transfer information on the disk. Note that the punctuation of the FILE statement is extremely tricky. The #-sign, colon and quotes are all required and in that order for a legal statement. The READ and WRITE statements have the form

```
200 READ#1: list
300 WRITE #1: list
```

The spaces, as usual, are unimportant, but the punctuation is required.

Now let us construct a program to write and reread a file from disk. We must first have a file on disk having the name we wish to use, in this case ABC. So, we type

```
STORE ABC.B :B
```

while running DEMON.

Then we run BASIC and define ABC to the BASIC directory by typing

```
DEFINE "ABC"
```

This enters the location of ABC in the BASIC directory permanently. It will be there whenever BASIC is restarted unless it is specifically deleted.

Now we begin typing our program. We must define the file as having a number within the program. Let us call it file number 1 by typing

```
100 FILE #1: "ABC"
```

We now wish to write some numbers onto disk. Let us do this by writing I^2 for I varying from 1 to 10.

```
110 FOR I = 1 TO 10
120 WRITE #1: I↑2
130 NEXT I
```

The above statements write I^2 onto disk where I varies from 1 to 10. The WRITE command simply says write the number I^2 into file number 1.

Now we wish to read these numbers back. We can only do this by going back to the beginning of the array. To reset the file "pointer" to the start of the file, we give the command

```
140 RESET #1: Ø
```

which says reset the file pointer of file number one to position zero, before the first datum. Now all we need to do is read the data back and print it. An exactly analogous loop is used.

```
150 FOR I = 1 TO 10
160 READ #1: X
170 PRINT X
180 NEXT I
```

The complete program is shown below including output.

```
*STO ABC.B:B

*RUN BASIC

READY
OLD "FTEST"

DEFINE "ABC"

READY
DIR

ABC       6144   U            0       0
READY
LIST

  "FTEST "
   100 FILE #1:"ABC"
   110 FOR I = 1 TO 10
   120 WRITE#1: I↑2
   130 NEXT I
   140 RESET #1:0
   150 FOR I=1 TO 10
   160 READ #1: X
   170 PRINT X;
   180 NEXT I
  9999 END
READY
RUN
 1  4  9  16  25  36  49  64  81  100
STOP
READY
DIR

ABC       6144   NUM          60       6
READY
```

Erasing BASIC Files

Once a BASIC file has been DEFINEd to BASIC, it is there permanently. BASIC will recognize it at any future time. However, BASIC space is limited and it is useful to remove files or reuse them. A BASIC file can be deleted from BASIC's internal directory by simply typing

```
DELETE "ABC"
```

A file can be erased and reused in BASIC by executing the instruction

```
100 SCRATCH #1
```

This means delete all information in the BASIC directory regarding a given file except its name. This makes the file type undefined and the length zero. Thus, certain files can be used over and over for temporary storage if desired. Once a file has been deleted from the BASIC directory, however, the information in it is lost. Redefining it to BASIC will produce an empty file of undefined type. Conversely, deleting a file in the DEMON directory that has not been deleted in BASIC

will cause an error message at the time BASIC is RUN, which will prevent BASIC from starting at all. This can be circumvented only be re-storing the file in DEMON before starting BASIC.

Terminal Format Files

The other principal type of file which BASIC uses is called terminal format. These files consist of information just as it would be typed on the teleprinter. The information may be string, or numeric, or some combination of both. It can be read and written from disk using the INPUT# and PRINT# statements. Note that the READ# and WRITE# statements are restricted to random access files and the INPUT# and PRINT# statements to terminal format files. This should prevent confusion in reading other people's programs.

The INPUT# and PRINT# statements work just like the READ# and WRITE# statements. For example,

```
INPUT #1: X,Y,Z
```

will read in the constants X, Y and Z from file number 1.

Remember, however, that the file number can be a variable such that

```
100 INPUT #A: X,Y,Z
```

is also legal if A has been evaluated prior to this statement. This is particularly useful in writing general programs, because BASIC has the convention that file Ø is the teleprinter itself. Thus if A is zero, data is taken from the Teletype and if any other number, from the appropriate file.

Preparing a List of Scores

Let us now suppose that we are going to have an assistant use a program for entering a list of student names and scores. This program will have to be extremely easy to use for a non-technical person and must produce the desired output onto disk so that it can be analyzed by some statistical analysis program. The program should

1. Print out instructions
2. Ask for data by type
3. Exit on condition that a student name QUIT is entered.

We will first define our file by name and number.

```
STORE SCORES.B :B
RUN BASIC
READY
DEFINE "SCORES"
100 FILE #1: "SCORES"
110 REM GET STUDENT NAMES AND GRADES
```

We will now accept a student name. Since Nicolet BASIC is restricted to 15 characters and since it assumes that data are separated by commas, we will have the user type in the student names last name followed by a comma followed by a first name, where each name is a different string constant. Then we will accept data from the keyboard, one score per line with three scores overall. If the last name entered is QUIT the program will stop. Thus, calling a student QUIT, calls it quits with the program.

```
140 INPUT A$,                    [NOTE COMMA
150 IF A$="QUIT" THEN 9999       [CALL IT QUITS
160 INPUT B$
170 FOR I = 1 TO 3
180 PRINT "Q";
190 INPUT G(I)
200 NEXT I
```

After getting both the last and first names and checking for QUIT, the program allows entry of three quiz scores. Here, since the program may be used by an inexperienced person, we print out a Q before each quiz score entry so that score can be distinguished from names.

Now, after each line of data has been entered, we wish to write out this data onto disk file "SCORES" by simply using a terminal format print statement

```
210 PRINT #1: A$, B$, G(1), G(2), G(3)
220 GO TO 140
```

This writes the data into file 1, which is defined as being "SCORES" in statement number 100.

The complete program, along with input is shown below.

```
LIST

   "GSCORE"
    100 FILE #1: "SCORES"
    110 REM GET STUDENT NAMES AND SCORES
    120 PRINT "ENTER NAME ";"LAST, FIRST ";"WITH SCORES ON ";
    130 PRINT "FOLLOWING LINES"
    140 INPUT A$,
    150 IF A$="QUIT" THEN 9999
    160 INPUT B$
    170 FOR I = 1 TO 3
    180 PRINT "Q";
    190 INPUT G(I)
    200 NEXT I
    210 PRINT #1: A$,B$, G(1),G(2),G(3)
    220 GO TO 140
   9999 END
READY
RUN
ENTER NAME LAST, FIRST WITH SCORES ON FOLLOWING LINES
?BIRD, BITCHY
Q?3
Q?6
Q?7
?RUTABAGA, VAUGHN
Q?3
Q?6
Q?1
?FARKLE, IGOR
Q?5
Q?7
Q?2
?QUIT
STOP
READY
```

Reading the Scores File

Now let us design a program to read this file and compute some simple averages. The actual average calculation is trivial, but printing out neat output will require some work as well. We start our program as before by defining a file number as a file name.

```
100 FILE #1: "SCORES"
```

Then, since we are going to average all the quizzes in the file, we must set our running sum to zero and our student counter to zero as well.

```
120 LET Q1=0
130 LET Q2=0
140 LET Q3=0
150 LET I=0
```

The variables Q1, Q2, and Q3 will be the sums of the scores of quizzes 1, 2 and 3 and the counter I will be the number of students whose scores have been read. Thus we need not know the length of the file when we start. Instead we will just count the entries as we read them in and keep averaging until the end of the file is found. We will also have our program print out a header and print out each line of scores as they are read in. This provides a check that the program is working properly as well as a neat listing of the students and their scores. Then we will simply add the quizzes to the running sum and count the number of entries by incrementing the counter I.

```
160 INPUT #1: A$, B$, K,L,M     [GET A NAME AND SCORE SET
165 PRINT A$, B$, K;L;M         [PRINT IT
170 LET Q1 = Q1 + K
180 LET Q2 = Q2 + L             [SUM SCORES
190 LET Q3 = Q3 + M
200 LET I = I + 1               [COUNT STUDENTS
```

The above code is the guts of the program, containing the reading of scores and names from a file and the summing of the scores. As yet, it has no terminus as we do not know how many students are in the list. This could have been specified at the outset and a FOR...NEXT loop set up, but it is more useful to simply let the program read until it runs out of data. A special form of the IF statement is used to test for the end of data in a particular file. This statement has the form

```
210 IF END #1 THEN 300   [GO TO 300 IF FILE DONE
220 GO TO 160
```

Statement 210 says "If there is no more data in file #1, go to 300." Statement 220 says "Otherwise go back to 160 and continue the loop." This IF END statement is of great value whenever the exact length of a file is unknown or inconvenient to find out.

The actual quiz averages will now be computed in the print statement where they are printed out. This great power of BASIC allows quite compact code, as any evaluated expression can be put in a print statement. So that a blank line will be printed for spacing after the student list, we begin with an empty PRINT statement and then print out the averages.

```
310 PRINT                          [BLANK LINE
320 PRINT "AVERAGES FOR ";"STUDENTS"
330 PRINT "QUIZ1="; Q1/I
340 PRINT "QUIZ2="; Q2/I
350 PRINT "QUIZ3="; Q3/I
360 PRINT "OVERALL AVG. =";(Q1+Q2+Q3)/(3*I)
```

The complete program, with output is shown below:

```
LIST

  "AVGSCR"
   100 FILE #1: "SCORES"  [DEFINE FILE TO PROGRAM
   110 REM ZERO CONSTANTS BEFORE STARTING AVERAGE
   120 LET Q1=0
   130 LET Q2=0
   140 LET Q3=0                  [QUIZ AVERAGES
   145 PRINT "STUDENT QUIZ";" LIST"
   150 LET I=0                   [STUDENT COUNTER
   160 INPUT #1: A$,B$,K,L,M
   165 PRINT A$,B$,K;L;M
   170 LET Q1=Q1+K
   180 LET Q2=Q2+L
   190 LET Q3=Q3+M
   200 LET I= I+1                [COUNT NUMBER OF STUDENTS
   210 IF END #1 THEN 300        [ CONTINUE UNTIL END OF FILE FOUND
   220 GO TO 160
   300 REM COMPUTE AVERAGES
   310 PRINT                        [NOTE EMPTY PRINT TYPES CR-LF
   320 PRINT "AVERAGES FOR ";"STUDENTS"
   330 PRINT "QUIZ1=";Q1/I
   340 PRINT "QUIZ2=";Q2/I
   350 PRINT "QUIZ3=";Q3/I
   360 PRINT "OVERALL AVG. =";(Q1+Q2+Q3)/(3*I)
  9999 END
READY
RUN
STUDENT QUIZ LIST
BIRD            BITCHY          3  6  7
RUTABAGA        VAUGHN          3  6  1
FARKLE          IGOR            5  7  2

AVERAGES FOR STUDENTS
QUIZ1= 3.66666
QUIZ2= 6.33333
QUIZ3= 3.33333
OVERALL AVG. = 4.44444

STOP
READY
```

The MEM Function

Nicolet BASIC provides the unique capability of accessing disk files in signal averager (integer) format. In other words, data obtained by the wired averager mode can be accessed and manipulated in BASIC. This is done using the MEM function. This function has two arguments; the first is the file number and the second the index of the data. The MEM function will then return the value of that word of the stored disk file. As before, the file must have a .B extension to be found by BASIC, even though it is signal averaged data.

The following code will allow one to access a file named "PMRFT.B" and integrate the points picked out previously by some cursor routine as decimal addresses 1413 through 1429. The data points are printed out for examination, although this is clearly unnecessary, except for reassurance.

```
READY
LIST

   "INTGRT"
    100 FILE #3: "PMRFT"
    110 LET A=0
    120 FOR I=1413 TO 1429
    125 LET X = MEM(3,I)
    127 PRINT X
    130 LET A=A+X
    140 NEXT I
    150 PRINT "INTEGRAL =";A
   9999 END
READY
RUN
 2886
 4375
 4353
 5402
 7764
 10333
 14139
 16439
 16552
 10134
 7506
 4954
 3955
 1768
 1040
 496
-281
INTEGRAL = 111815

STOP
READY
```

It should be noted that because the MEM function swaps out part of BASIC to read in the file to be accessed, the MEM function cannot appear in a statement containing other disk input output functions or commands.

CHAPTER 3. Specifications of NICOLET BASIC

I. Constants

In BASIC, there are two types of data: numbers and strings. A number is always represented in floating-point format and must conform to the rules set by the Nicolet Floating Point Package - 1972. A floating-point number consists of from 1 to 11 digits, possibly containing a decimal point and optionally followed by the letter E and a power of ten.

In most parts of BASIC a string constant consists of 0 to 15 characters enclosed in quotes ("), e.g., "ABC". Under certain conditions, the quotes may be omitted in typing in data (see below). Characters must belong to the stripped ASCII set used by the 1080 assembler (Appendix II).

The two data types may not be mixed. An expression such as 1.5 + "ABC" is illegal.

II. Variable and Array Names

Simple numeric variable names in BASIC are restricted to a single letter, or a letter followed by a digit, e.g., A or A1. Simple string variable names consist of a letter followed by a dollar sign, or a letter, a digit and a dollar sign, e.g., A$ or A1$.

Numeric array names may only be a single letter, while string array names consist of a single letter followed by a dollar sign. Unlike Dartmouth BASIC, the same name may not be used both as an array name and a simple variable name in the same program. Arrays may have one or two dimensions. Singly-dimensioned arrays will be referred to as vectors; those with two dimensions are matrices. Internally, arrays are stored by rows, a fact which will be significant for Matrix I/O (section VI-C). Array references are of the form A(X) or A(X,Y) where X and Y may be any arithmetic expressions. The subscripts will be the integer part of such expressions. Note that, since all arithmetic is done in floating point, the user must be careful that roundoff errors do not produce erroneous subscripts. If a computation should produce the result 4.0, it is entirely possible that it might result in 3.99999+, which would give a subscript value of 3, not 4. The value of a subscript must be greater than zero and less than or equal to the corresponding dimension of the array.

III. Expressions

Expressions in BASIC are either string or arithmetic expressions; the two modes may not be mixed.

A number, numeric variable (simple or subscripted) or numeric function reference is an arithmetic expression. More complex expressions are formed by the following rule: if X and Y are arithmetic expressions then so are:

(X) X+Y X-Y X*Y X/Y X↑Y -X

The operator "*" stands for multiplication, the operator "↑" for exponentiation.

A string constant, string variable (simple or subscripted) or string function is a string expression. If A$ and B$ are string expressions, then so are A$ & B$. The operator "&" is the string concatenate operator and is the only string operator permitted. For example, if A$ = "ABC" and B$ = "DEF" then A$ & B$ = "ABCDEF". The result of a string concatenation may not exceed 15 characters.

IV. Statements

This section consists of a list of statements available in BASIC, with a description of each. Some general rules apply to all statements. Each statement must have a statement number in the range 1 to 99999. When a program is read in by the compiler, the statements are ordered by statement number, regardless of the sequence in which they are read. Throughout the following examples, the statement number 100 is arbitrarily used. A statement may be 1 to 72 characters in length. BASIC interprets everything up to the carriage return or up to the character "[" (SHIFT-K). Anything between "[" and the carriage return is treated as a comment. "[" may appear in a string constant or an input string.

A. LET statement. This is the fundamental statement of BASIC. It corresponds to the assignment statement of FORTRAN. The general form is:

```
100 LET v = e
```

where v is a variable (simple or subscripted) and e is an expression whose type (string or numeric) must match that of v. The expression is evaluated and its value is assigned to v.

B. DIM statement. This is used to declare arrays and to assign dimensions to them. The form is

```
100 DIM     v (e),  v (e, e)
```

where v denotes a simple variable name (string or numeric) and e denotes a numeric expression. The integer part of the expression is used and its value must be ≥ 1. The maximum number of dimensions is 2. DIM statements may appear anywhere in the program. Previously dimensioned arrays may be re-dimensioned at any time,

although the space required by the new declaration may not exceed the space allocated originally. An interesting feature of BASIC is the ability to dimension arrays implicitly and automatically. If a subscripted variable is used which has not been declared, it is automatically dimensioned (10) or (10,10), depending on the number of subscripts used in the reference. This feature does not apply in MAT statements.

C. GO TO statement. This statement allows deviation from normal sequential execution of the program. The form is

100 GO TO s

where s is a statement number. The next statement executed will be the one numbered s.

D. GOSUB and RETURN statements. These statements allow the user to call and return from internal subroutines. The forms are

100 GOSUB s

and 200 RETURN

Control is transferred to the statement numbered s. When the next RETURN statement is reached, control is transferred back to the statement following the GOSUB. GOSUB . . . RETURN statements may be nested up to 24 deep. Recursive GOSUB calls are permitted.

E. ON . . . GO TO and ON . . . GOSUB. These statements are similar to the computed GO TO in FORTRAN. The form is

100 ON e GO TO $S_1, S_2, \ldots S_n$

where e is a numeric expression and the S_i are statement numbers. The expression is evaluated and its integer part (say j) is obtained. If $1 \leq j \leq n$, control is transferred to statement S_j. If $j < 1$ or $> n$, an error message is given and execution terminates. Thus, ON A+1 GO TO 200, 300, 400 will transfer control to 200 if A = 0, 300 if A = 1 and 400 if A = 2. The ON. . . GOSUB statement is exactly analogous.

F. IF statement. This statement provides the ability to execute a branch or not, depending upon whether a specified condition is true.

100 IF e_1 rel e_2 THEN S

where e_1 and e_2 are expressions, S is a statement number, and rel is one of the following relations:

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

If the specified condition holds between the two expressions, control is transferred to statement S. Otherwise, execution continues sequentially. The expressions may be either string or numeric, but they may not be mixed in the same statement. In the case of strings, they are ordered alphabetically according to the table of packed 6-bit codes in Appendix II. That is, "ABC" < "BBC". If the strings are not of equal length, and are identical up to the end of the shorter string, the shorter one is the lesser of the two, e.g., "AB" < "ABC". Trailing blanks are meaningful: "AB" < "AB ".

G. FOR and NEXT statements. These statements are similar to the FORTRAN DO statement. The form of the FOR statement is

100 FOR v = e_i TO e_f STEP e_s

where v is the "running variable." It must be a simple numeric variable. e_i, e_f, and e_s are numeric expressions. The form of the NEXT statement is

200 NEXT v

where v is again a simple numeric variable, corresponding to the variable in the FOR statement.

The FOR and NEXT statements form a "bracket" around other statements which are executed repeatedly. When the FOR statement is encountered, the value of e_i is assigned to the running variable, v. When the NEXT statement is encountered, the value of the STEP expression, e_s, is algebraically added to v. The resulting value is compared with e_f. If e_s is negative (positive), and the result is less than (greater than) e_f, the statement following the NEXT statement is executed. Otherwise, control is transferred to the statement following the FOR statement and the loop is executed again. If the conditions in the FOR statement are "impossible," such as FOR I = 1 TO 4 STEP -1, the loop is not executed at all. The STEP e_s portion of the statement is optional. If not present, the step size is taken to be +1.

FOR loops may be nested up to 8 deep. The statements in the FOR loop, including the FOR and NEXT statements, are called the range of the loop. If two or

more loops are nested, the range of each "inner" loop must be contained within the range of each loop "outside" it. Using a bracket to denote the range of a loop, the examples illustrate legal and illegal nesting.

Jumps into the range of a FOR loop from outside that range may yield unpredictable results. However, it is legal to jump out of a loop, then back into it, or to execute GOSUB statements within it. The user should be prepared for strange behavior if the running variable is altered before returning to the loop.

Upon normal exit from a loop, the running variable retains the value which caused exit. For example,

```
100 FOR I = 1 TO 10
```

When the loop is exhausted I has the value 11.

H. RANDOMIZE statement. This statement is used to initialize the random-number function RND. The form is

```
100 RANDOMIZE e
```

where e is a numeric expression. The mantissa of the expression value replaces the current value in RND, thus changing the sequence generated. If RANDOMIZE is not used, RND uses the mantissa of π as its starting value.

I. STOP statement. This statement stops execution of the program, prints "STOP" on the printer, and transfers control back to the BASIC processor.

J. END statement. The END statement terminates the program text. Thus, it must have the highest statement number in the program. If encountered in execution, the END statement also functions as a STOP statement. It is therefore generally useful to give the END statement number 99999.

K. DEF statement. This allows the user to DEFine statement functions. The function name must be of the form FNα (numeric functions) or FNα\$ (string functions), where α is a letter. The form is

$$100\ \text{DEF FNA}\ (a_1, a_2, \ldots a_n) = e \qquad (1 \leq n \leq 5)$$

where the a_i are the function arguments and e is an expression of the same type as the function name. A function reference may appear in any arithmetic expression except the right-hand side of its own DEF statement. If this condition is violated, the parsing stack will overflow and error SOV will be given.

L. REM statement. This allows insertion of comments into the BASIC text. It is ignored during execution, although a GO TO, GOSUB, IF, or ON statement may reference it. The form is

```
100 REM any text.
```

V. Input-Output

BASIC input-output statements are sufficiently complex to warrant a separate section.

A. I/O units. In order to make clear some of the succeeding sections, a discussion of input-output units is appropriate at this point. In general, data may be read from the keyboard, from disk files, or from DATA statements within the program itself. Data may be written on the printer or disk files. Disk files are further divided into terminal-format (sequential) files and random-access files. Terminal-format files may contain records of variable length, much as one types variable-lenigth records on the Teletype or terminal. These records are stored on the disk as packed 6-bit characters, three per word. The end of each record is designated by the code 77_8. This particular packing and terminating scheme makes "←" back arrow character illegal. For output, the maximum record size can be set by the user, up to 72 characters for the terminal and 1535 characters for disk files. This limit can be changed at any time and does not affect input operations. Random-access files which are not empty have a fixed record size, thus allowing calculation of the position of a desired record. Random-access files are either numeric or string. In the case of numeric files, the record size is always 6 characters (two words). For empty string files, the user may set the record size. If the user does not set the record size, the processor sets it at 72 for terminal-format files (including the terminal) and 15 for random-access string files. Note that in a random access string file all strings are the same length, regardless of their length in the program prior to writing them in the file. If a string of length n is written into a string file with margin m and later read back, the string read will be of length m, with blank fill if necessary.

Each disk file has a file pointer which points to the next record to be read or written. This pointer is always set to zero by BASIC when a file is first referenced by the FILE statement. When an empty file is referenced by the FILE statement, its type is set to "undefined." When it is later used for I/O, its type is set to correspond to the kind of statement being executed.

B. FILE statement. This statement allows the user to equate a file number, which will be used to identify the file in input-output statements, with the name of a file on the disk. This name must exist in the BASIC file directory as well as in the DEMON directory. The DEFINE statement (Chapters 2, 4) is used to enter the file name into the BASIC directory. The form of the FILE statement is:

100 FILE #e_1:e_2

where e_1 is a numeric expression whose integer part is the desired file number and e_2 is a string expression which is the desired file name. The integer part of e_1 must meet the condition $0 < [e_1] < 1024$. The number sign and colon are required punctuation.

C. PRINT statement. This statement is used to write data on terminal-format files, including the terminal itself. There are two forms:

100 PRINT list

200 PRINT #e: list

where e is a numeric expression designating the desired output file and "list" is a sequence of expressions whose values are to be printed. The items of the list must be separated by commas or semi-colons. The first form is used to print on the terminal, the second for disk files. However, a file number of zero also designates the terminal.

If the list items are separated by commas each is printed starting at the beginning of a 15-column "zone." These zones begin in columns 1, 16, etc. If a list item is followed by a semi-colon, it is printed in "close-packed" format; that is, a numeric item is followed by a single blank, while a string item has no blanks following it.

Each numeric item is preceded by a blank, if positive, or a minus sign, if negative. The format of the number itself is determined by its value, v, as follows:

1. If v is an integer and $|v| < 2^{25}$, v is printed in integer format with no decimal point.

2. If $0.1 \leq |v| < 10^7$ and v is not an integer, it is printed in fixed point format with 6 significant digits and a decimal point, e.g., 1.23456.

3. Otherwise, v is printed in scientific notation format with 6 significant digits, a decimal point and a power of 10. For example, 1.23456E10, which means 1.23456×10^{10}.

Normally, at the end of a PRINT statement the carriage is returned, if the terminal is being used, or the CR character is entered and the record is written out on the disk, if a file is being used. No more output can be added to such a record. But if the print list ends with a comma or semi-colon, the record is not terminated and the next PRINT statement will continue to write in the same record. For example, the sequence

```
100 FOR I = 1 TO 3

200 PRINT I,

300 NEXT I
```

will produce

```
1             2             3
```

A PRINT statement writes as many records as are necessary to print the entire list. If the list is empty, an empty record is written. The length of a record is determined by the margin of the file (see MARGIN statement below).

D. INPUT statement. This statement is used to read data from terminal-format files and the terminal. There are two forms:

```
100 INPUT list

200 INPUT #e: list
```

where e is a numeric expression designating the desired input file (file 0 is the terminal), and "list" is a sequence of variable names (simple or subscripted) to which the values read are to be assigned. The items in the list must be separated by commas.

Each item in the list must correspond in type (string or numeric) with the value read. The type of the input datum is determined in the following manner:

1. If the first non-blank character is a digit, decimal point, or sign, the datum is assumed to be integer.

2. If the first non-blank character is a quote, the datum is assumed to be a string enclosed in quotes.

3. If the first non-blank character is none of the above, the datum is considered to be an unquoted string. Such a string may be up to 15 characters long. It terminates at the 15th character, the first comma encountered, or the end of the record.

Normally, each request for input from a terminal-format file will take data from a new record. If the terminal is being used, a carriage return, line feed, and question mark will be printed. Any data remaining in the previous record will be lost. However, if the preceding operation on that file was an INPUT statement ending with a comma, the new INPUT statement begins by reading any data remaining in the record before going on to the next. If an INPUT statement references the terminal, and the last operation on the terminal was a PRINT ending with a comma, the carriage return and line feed are omitted and a question mark is printed. An INPUT statement will read as many records as are necessary to satisfy the list. On the terminal, the CR-LF-? sequence is printed for each new record needed. The LINPUT statement reads data from the input device, one datum per record. From the teleprinter, this means one number or string per line. Further data on the same line are ignored.

E. LINPUT statement. This statement can be used to read unquoted strings, possibly containing quotes, commas, or leading blanks, digits, signs, or decimal points. The LINPUT statement uses the terminal or terminal-format files. Each string is taken from the beginning of a new record and consists of the first 15 characters in the record, or the entire record if it is less than 15 characters long. The form is

```
100 LINPUT list

200 LINPUT #e:list
```

where e is a numeric expression designating a file (0 is the terminal) and "list" is a sequence of variables (simple or subscripted) separated by commas.

F. WRITE statement. This statement is used only to write random-access files. Its form is

```
100 WRITE #e:list
```

where e is a numeric expression whose integer part (> 0) is the file to be written and "list" is a sequence of expressions, separated by commas, whose values are to be written. All expressions in the list must be of the same type and must correspond with the file type. Each item in the list is written into a single fixed-length record

on the disk. For numeric items the record consists of the two-word internal representation as described in the Nicolet Floating Point Package manual. For string items the record consists of 6-bit ASCII characters packed three per word. All strings are of the same length: the margin of the file. If a string is longer than the margin, it is truncated; if shorter, it is blank filled.

Each item in the list is written starting at the current value of the file pointer and the pointer is advanced accordingly.

G. READ statement. This statement has two forms and two uses. The first is

```
100 READ list
```

where "list" is a sequence of variables (simple or subscripted) separated by commas. This form of the READ statement takes data from DATA statements in the program itself. The DATA statement has the form

```
100 DATA data list
```

where "data list" is a sequence of values following the rules described in Section D. The program may have as many DATA statements as desired. When the values in a DATA statement are exhausted, BASIC moves on to the next, if any. Separate pointers are maintained for numeric and string data and the READ statement always takes the next value of the appropriate type. If no more data remain, an error message is given and execution terminates.

The second form of the READ statement is

```
100 READ #e: list
```

where e is a numeric expression whose integer part (> 0) designates a random-access file from which data are to be read. All items in the list must be of the same type and must agree with the type of the file to be read. Values are read beginning with the one indicated by the current value of the file pointer. The pointer is updated as each value is read.

H. MARGIN statement. This statement is used to set the maximum record size for output on the terminal, terminal-format files, or random-access string files. The form is

```
100 MARGIN e_m

101 MARGIN #e_1:e_m
```

where e_m is a numeric expression whose integer part (> 0) is the desired record size in characters. The margin of a terminal-format file may be changed at any time. The margin of a random-access string file may be changed only if the file is empty. The margin of a random-access numeric file may never be changed. It is always 6. For the terminal the maximum margin is 72; for terminal-format files it is 1535; for random-access string files it is 15.

I. SCRATCH statement. This statement sets all file pointers and the margin to zero, thus erasing the file. It also sets the file type to "undefined." The form of the statement is

```
100 SCRATCH #e
```

where e is a numeric expression whose integer part (> 0) is the desired file number.

J. RESET statement. This statement has three forms. The first is

```
100 RESET
```

This causes the DATA statement pointers to be reset to the beginning of the first DATA statement.

The second form is

```
100 RESET #e
```

where the integer part of the numeric expression, e, is the number of the desired terminal-format file. The file pointer of the designated file is reset to zero, in effect performing a "rewind."

The third form is

```
100 RESET #e1:e2
```

where the integer part (> 0) of e, is the desired random-access file and the integer part of e_2 must designate a record number, not a character position. The file pointer is set to this position. Thus record number [e_2] will be the next record read or written.

K. IF END and IF MORE. These statements allow the user to test for the end of a terminal-format file. The form is

100 IF END #e THEN S

120 IF MORE #e THEN S

where e is the usual file number (> 0) and S is the statement to go to if the indicated condition is true.

VI. MAT Statements

BASIC has the extremely powerful ability to manipulate entire matrices in a single statement. A description of the available statements follows.

A. Matrix arithmetic

100 MAT A = B

120 MAT A = (e) * B

140 MAT A = B + C

160 MAT A = B - C

180 MAT A = B * C

In statements 100 and 120, A and B must both be matrices or both vectors. A is re-dimensioned, if enough space has been allocated for A, to be the same as B. In statement 120 each element of B is multiplied by the value of the expression e, which must be enclosed in parentheses. In statements 140 and 160, B and C must have identical dimensions and A is re-dimensioned to agree with them, if possible. In statement 180 the dimensions of B and C must be of the form m x k for B and k x n for C. A will be re-dimensioned m x n, if possible. For matrix multiplication, a vector can be considered either 1 x n or n x 1, depending upon its position on the right-hand side. If either B or C is a vector, A must also be a vector. Mathematically speaking if B and C are both vectors A is a scalar. BASIC, however, requires that A be a vector. The same matrix may not appear on both sides of the equation in matrix multiplication.

B. Matrix functions

1. 100 MAT A = CON(e_1, e_2)

This statement re-dimensions A to be e_1 x e_2 and sets all its elements to +1.0. If the "(e_1, e_2)" part is omitted, no re-dimensioning occurs. Dartmouth BASIC dimensions all matrices so that they have a zero column and zero row. This feature is seldom of value and since it greatly increases storage requirements, it has been omitted from Nicolet BASIC.

2. 100 MAT A = ZER (e_1, e_2)

This is the same as CON, except that all elements are set to zero.

3. 100 MAT A = IDN (e_1, e_2)

This is the same as CON, except that A is set to the identity matrix. If re-dimensioning is indicated, we must have $e_1 = e_2$. If no re-dimensioning is indicated, A must be square.

4. 100 MAT A = TRN(B)

This function sets A to be the transpose of B, re-dimensioning A if necessary. A = TRN(A) is illegal.

5. 100 MAT A = INV(B)

This sets A to be the inverse of B. A is re-dimensioned, if possible. A and B must be square. MAT A = INV(A) is legal. If the matrix to be inverted is singular, no diagnostic is given. The only indication of singularity is that DET (determinant) is set to zero. In this case, the matrix on the left side will contain garbage.

6. MAT A$ = NUL$($e_1$, e_2)

This sets each element of the string matrix A$ to the null string, re-dimensioning if possible and if indicated.

C. Matrix input-output

BASIC provides the ability to read or write entire matrices with a single statement. Each of the I/O statements discussed in Section V is represented in matrix I/O. In the following examples, items enclosed in the symbols < > are optional. Except as noted below the statements are the same as their non-matrix counterparts. Statements available are:

100 MAT INPUT <#e:> list

120 MAT LINPUT <#e:> list

140 MAT READ <#e:> list

160 MAT PRINT <#e:> list

180 MAT WRITE # e : list

where "# e :" is a file declaration as discussed in Section V, and "list" is now a sequence of matrix or vector names, separated by commas or, in the case of MAT PRINT, by semi-colons or commas.

1. MAT INPUT is the same as the non-matrix form except that each matrix in the list begins taking data from a new record. If the terminal is being used, "CR-LF-?" will be printed for each new matrix.

2. For MAT PRINT, the punctuation following the array determines the print format: a comma results in normal zone printing; a semi-colon results in close-packed format. For the last matrix in the list, if there is no punctuation following the name, zone format results. In the case of vectors, they are printed in row format, with zone spacing if followed by a comma, and in close-packed row format if followed by a semi-colon. If the last item in the list is a vector and there is no following punctuation, it is printed in column format, one element per record. In printing matrices, each row begins a new record.

VII. Functions

The following functions are available in BASIC:

A. ABS(e) returns the absolute value of the numeric expression e.

B. ATN(e) returns the arctangent (in radians) of the numeric expression e.

C. COS(e) returns the cosine of the numeric expression e. e must be in radians.

D. COT(e) returns the cotangent of the numeric expression e. e must be in radians.

E. DET, which has no arguments, returns the determinant of the last matrix inverted. If the matrix was singular, or if no matrix inversion has been done, DET returns zero.

F. EXP(e) returns the exponential function of the numeric expression e.

G. INT(e) returns the largest integer which is less than or equal to the numeric expression e.

H. LEN(e$) returns the number of characters in the string expression e$.

I. LOC(e) returns the current value of the file pointer, in characters, for the random-access file designated by the numeric expression e.

J. LOF(e) returns the character index of the highest-numbered record written in the random-access file designated by the numeric expression e.

K. LOG(e) returns the natural logarithm of the numeric expression e.

L. MAR(e) returns the current value (in characters) of the margin of the file designated by the numeric expression e.

M. MEM(e_1, e_2) returns the value of a word in data memory. e_1 is a numeric expression designating a disk file into which data memory was written prior to running BASIC. e_2 is a numeric expression designating the index of the desired item. The first item in the file has index 0. The indicated item is converted to standard floating-point format. MEM is available only in disk systems and may not be called in a disk output list.

N. MOD(e_1, e_2) returns the value of e_1 modulo e_2, where e_1 and e_2 are numeric expressions.

O. RND, which has no arguments, returns the next in a sequence of pseudo-random numbers. $0 < RND < 1$.

P. SGN(e) returns the value +1, 0, or -1, depending upon whether the numeric expression e is positive, zero, or negative.

Q. SIN(e) returns the sine of the numeric expression e. e must be in radians.

R. SQR(e) returns the square root of the numeric expression e.

S. TAB(e) causes the terminal to space to the column designated by the numeric expression e. e must be greater than the current column position of the terminal.

T. TAN(e) returns the tangent of the numeric expression e. e must be in radians.

U. USER(e_1, e_2) allows the user to write his own machine-language function. BASIC uses the following calling sequence in calling USER:

```
JMS USERF        /USERF = 106630
```

The values of e_1 and e_2 are contained in FAC and FAR, respectively. The result of the function should be left in FAC and the return is to the location following the JMS.

One possible loading sequence for getting USERF into core is:

```
*RUN BASIC
 READY     (printed by BASIC)
 BYE       (typed by user to return to DEMON)
*LOAD USERF  (assumed to be stored on disk)
*GO 0        (return to BASIC)
```

For paper-tape systems, simply load the BASIC tapes, then load the USERF tape and go to location 0.

One powerful possibility for USERF in disk systems is to load an overlay into core, execute it, and return through USERF. The user must save and restore the overlay area. The facilities of DEMON, including the directory functions and disk I/O routines are available for use. Although non-disk systems cannot make use of overlays, the core areas 100740-101272 and 107500-107777 are available for use. These areas are used for disk file tables and routines and are not used in non-disk systems.

Note that USER can actually be a variety of functions since e_1 and e_2 can be indices of one of several desired routines. The disk user function space is limited to 64 locations from 106630 - 106727. Longer routines can be used only by having part of the BASIC program written out onto disk and the longer program swapped in its place using the DIRFUN and IOSUPER handlers from DEMON. The portion of BASIC swapped out must be restored before returning from the USER function.

CHAPTER 4. Operating BASIC

This chapter gives a summary of the commands available in BASIC and describes program input and editing.

A. Commands available in both disk and tape systems:

1. NEW "name"

This establishes the name of a new program. It also initializes all pointers, thus erasing the old program, if any. BASIC responds with a carriage return and line feed.

2. RENAME "name"

The name of the current program is changed to "name". BASIC responds with CR-LF.

3. LIST

The current program, if any, is listed on the terminal. The listing is preceded by the program name. When the listing is finished, BASIC types "READY." Listing can be terminated by typing "CTRL/Q."

4. RUN causes the current program to be executed. Execution can be terminated by typing CTRL/Q. This aborts execution at the beginning of the next statement.

5. TAPE causes the high-speed reader to be the input device, replacing the terminal.

6. KEY causes the terminal to be the input device. BASIC responds with "READY."

7. PUNCH causes the text of the current program to be punched on the high-speed punch. The text is preceded by

NEW "name"

and followed by a "KEY" directive.

B. Commands available only in disk systems:

1. SAVE causes BASIC to write out the text of the current program to the disk. The text is written into a file named "pgm-name.B" If such a

file already exists, DEMON types "DELETE:". If you answer "Y" the new version replaces the old.

2. UNSAVE "name" causes file "name.B" to be deleted from the disk. DEMON types "DELETE:". If you answer "Y" the file will be deleted.

3. OLD "name" causes the previously saved program "name" to be loaded from "name.B" on the disk. The program is ready for execution, editing, etc.

4. DEFINE "name" causes file "name" to be entered in the BASIC directory. The track address and file length are placed in the directory. The file type, margin, and amount written in the file are all set to zero. A file called "name.B" must exist on the disk when the DEFINE directive is entered.

5. DELETE "name" removes the file "name" from the BASIC directory. The file remains on the disk.

6. DIR produces a listing of the BASIC directory in the following format:

name size type contents margin

where size is the capacity of the file, in characters,
type is U for undefined, TER for terminal format, NUM for random-access numeric, STR for random-access string,
content is the number of characters written into a terminal-format file, or the highest character address written in a random-access file,
margin is the current margin in characters.

7. BYE returns control to DEMON.

C. Entering and editing a program.

Program entry and editing are simple. It is necessary to establish a program name using the NEW directive or to read in an old program using the OLD directive. You may then enter BASIC statements. If the statement number already exists in the program, the new statement replaces the old one. If no existing statement has the same number, the new statement is inserted in the text. Its position is determined by its statement number (statements are ordered in increasing statement number sequence). If the new statement is empty, i.e., consists only of a statement number, and its number already exists, the old statement is deleted. If no such number exists, the statement is ignored.

When first entering a program, allowing gaps of 10 between statement numbers will simplify editing, correcting and debugging.

When entering a statement or directive, if you want to backspace and delete erroneous characters, simply type "RUBOUT" the appropriate number of times. Each time "RUBOUT" is typed, BASIC prints "\" and deletes another character. If you backspace to the beginning of the line, further RUBOUT characters are ignored. To void the entire line type "CTRL/O."

D. Establishing BASIC on the disk.

For those with a disk system, the following procedure will establish BASIC on the disk:

1. Load the binary tape marked BASIC. Then type

   STO BASIC 0-1777;0:P

2. Load the binary tapes marked BASIC1, BASIC2, and BASDIR.

   Then type

   STO BASIC1 0-7577:P
   STO BASIC2 102000-107777:P
   STO BASDIR 100500-101677:P

   BASIC is now ready for execution.

   To execute, type

   RUN BASIC

or LOAD BASIC
   GO

   When BASIC is executed, it performs the following tasks:

   a. Loads BASDIR and fills in the current track address and size of each file in the BASIC directory. BASIC will search up to four disks.

   b. Loads BASIC1 and BASIC2, overlaying itself, and jumps to location 0.

If BASIC1, BASIC2, BASDIR, or any of the files in the directory does not exist, BASIC types

XXXXXX IS UNDEFINED. EXITING TO DEMON.

Establish the missing file on disk and try again.

E. Executing BASIC in a paper tape system.

Using the binary loader, load tapes BASIC1, BASIC2, and BASDIR. Begin execution at location 0.

APPENDIX I. Error Messages

The following errors result in an error message. Execution continues.

ARI — Arithmetic overflow or underflow

DIM — Syntax error in DIM statement. The offending array declaration is ignored.

DLZ — In a DIM statement a dimension is ≤ 0. The offending array declaration is ignored.

DMN — In a DIM statement, a dimension is too large or a re-dimensioned array is larger than the space previously allocated. The declaration is ignored.

FER — The FPP error flag was set during an intrinsic function call.

MEM — In a MEM call, the item index was < 0 or $>$ the file length. A value of zero is returned.

NUM — Error in numeric constant (overflow or underflow).

SCC — In a string concatenation, the result is longer than 15 characters. The first 15 characters are used.

STL — String constant is longer than 15 characters. The first 15 characters are used.

STW — Erroneous string length in output statement. If length was < 0, the null string is used; if > 15, the first 15 characters are used.

TAB — Argument of TAB function was less than current line position or greater than the margin. The TAB call is ignored.

TMD — Too many dimensions in array declaration (DIM statement). The offending declaration is ignored.

The following errors terminate execution of the statement in which they occur:

FNM — Error in file name in FILE statement

RNZ — Error in RANDOMIZE numeric expression.

TMF	Too many files (more than seven) have been declared in FILE statements. The FILE statement is ignored.
TRM	Illegal statement termination, e.g., GO TO 100 X = Y
UDF	Undefined file. This error aborts the statement if it occurs in a FILE statement.

The following errors terminate execution of the program:

ARG	Function parameters do not match formal parameters in number or type.
ARN	Array name used as simple variable.
CHR	Illegal character encountered.
DCP	Decimal point out of place.
DFS	Syntax error in left side of DEF statement.
DFO	Disk file overflow.
DIG	A letter is followed by two consecutive digits, e.g., A12. This usually means an operator has been omitted.
DLR	Dollar sign out of place.
EOF	Tried to read past the end of the file.
EDT	End of data in DATA statements.
EST	Erroneous statement type (a legal statement type is LET, PRINT, etc.)
FEQ	Missing equals sign in DEF statement.
FLN	Error in file number.
FNS	Function call not permitted in this statement (e.g., MEM call in disk I/O statement).
FPA	Error in formal parameter list.
FRM	Non-numeric expression in FOR statement.
FRN	FOR statement nesting error.

FRQ	Missing element in FOR statement.
FRV	FOR running variable missing or not simple numeric variable.
FSO	FOR stack overflow or underflow. This means that a FOR nest more than 8 deep has been found or that the FOR and NEXT statements are not properly prepared.
FTP	Error in file type.
GSS	Error in GOSUB statement number.
GST	GOSUB stack overflow (GOSUB nest is more than 24 deep).
GTS	Error in GO TO statement number.
IFC	Error in IF conditions, e.g., more than two conditions or same condition appears twice.
IFS	Syntax error in IF statement.
LFT	Syntax error in input list item or left side of LET statement.
MDM	MAT dimension error.
MGN	MARGIN value ≤ 0 or too large for file or unit.
MRN	Attempt to change margin on a random-access numeric file.
MST	MAT not followed by array name or I/O command.
MSF	In MAT statement attempted re-dimensioning requires more space than originally allocated for the array.
MTS	Syntax error in MAT statement.
MXI	Data type did not match input list item type.
MXM	Mixed mode.
NDM	Number of subscripts does not agree with number of dimensions.
NDT	No DATA statements in program.
NPD	Illegal construction in expression, e.g., an array reference A(I,J,K)
NRL	Illegal construction in expression, e.g., B + -C.

ONG	GOSUB or GOTO missing in ON statement.
ONS	Erroneous statement list in ON statement.
ONX	ON branch index out of range.
QUO	Missing closing quote in string constant.
RET	GOSUB stack underflow, i.e., unmatched RETURN statement.
RFZ	Attempt to use file 0 for random access.
RST	RESET statement has bad record number.
SBS	Subscript value ≤ 0 or greater than dimension.
SCR	Syntax error in SCRATCH statement.
SOV	Parsing stack overflow.
STF	Symbol table is full. This means that the program text plus variable storage exceeds the allocated space.
TRM	Illegal statement termination.
UDF	Undefined file in statement other than FILE.
UDS	Statement referenced by GOTO, GOSUB, ON, or IF statement does not exist.
UDV	Undefined variable.
USE	The function USER was called but none was loaded.
WRS	Syntax error in WRITE list.

The following errors are detected in directive processing, text entry and editing, or preparation for execution following a RUN directive:

BDF	BASIC file directory is full. It will hold 28 entries. (DEFINE)
CLS	Error flag set after DIRFUN CLOSE call. (SAVE and UNSAVE)
DFL	Disk is full (SAVE).
DIR	Statement number or directive is missing or erroneous.

DNM	Missing name or file already defined. (DEFINE or DELETE)
END	Program has no END statement. (RUN)
MDF	Multiply - defined user function. (RUN)
NFN	Function name missing in DEF statement. (RUN)
NPG	No active program or program not found. (OLD, RUN, LIST)
STN	Statement number < 1 or > 99999.
UDF	Undefined file.

APPENDIX II. Character Codes.

The following characters are available for strings in BASIC:

Character	Code	Packed 6 Bit	Character	Code	Packed 6 Bit
A	301	41	!	241	01
B	302	42	"	242	02
C	303	43	#	243	03
D	304	44	$	244	04
E	305	45	%	245	05
F	306	46	&	246	06
G	307	47	'	247	07
H	310	50	(	250	10
I	311	51	)	251	11
J	312	52	*	252	12
K	313	53	+	253	13
L	314	54	,	254	14
M	315	55	-	255	15
N	316	56	.	256	16
O	317	57	/	257	17
P	320	60	:	272	32
Q	321	61	;	273	33
R	322	62	<	274	34
S	323	63	=	275	35
T	324	64	>	276	36
U	325	65	?	277	37
V	326	66	@	300	40
W	327	67	[	333	73
X	330	70	\	334	74
Y	331	71	]	335	75
Z	332	72	↑	336	76
0	260	20	Return	215	77
1	261	21	Space	240	00
2	262	22			
3	263	23			
4	264	24			
5	265	25			
6	266	26			
7	267	27			
8	270	30			
9	271	31			

INDEX