PRINTING RECORD

Printing	Notes
08/76	Original Printing
09/77	Revision A. The following pages have been revised: vi, 2–9, 2–11, 2–27,
	4–19, 4–25, 7–10, A–8, A–10, A–12, A–13, A–14, B–1,
	B–2, B–3, B–14, B –17, C–13, D–1, D–6.
	The following pages have been added: 4–53, 4–54, 4–55,
	4–56, 4–57, C–14
03/79	Revision B. The following pages have been revised: 2–27, 4–53, 4–54,
	4–55, 4–56, 4–57, 7–10, A–12, B–1, C–14, D–2, D–6.
01/80	Revision C. The following pages have been revised: 1–4, 1–6, 2–2, 2–3,
	2–6, 2–7, 2–12, 2–31, 3–1, 3–5, 3–7, 4–1, 4–2, 4–3,
	4–4, 4–5, 4–7, 4–8, 4–9, 4–11, 4–14, 4–17, 4–20, 4–21,
	4–26, 4–30, 4–31, 4–32, 4–36, 4–39, 4–41, 4–44, 4–46,
	4–50, 4–51, 4–53, 4–54, 5–1, 5–6, 5–10, 5–11, 5–13,
	5–14, 5–16, 7–3, 7–8, 7–9, 7–10, 7–11, 7–12, 7–13, 7–14,
	A–1, A–6, A–8, A–10, A–16, A–18, B–1, B–5, B–6, B–7,
	B–8, B–12, B–13, B–14, B–15, B–16, B–19, B–20, C–1,
	C–5, C–11, C–12, D–2, D–5, and added page: A–10a.
05/81	Revision D. The following pages have been revised: 1–4, 1–5, 2–6, 2–8,
	2–9, 2–11, 2–18, 3–1, 4–8, 4–16, 4–17, 4–19, 4–24, 4–25,
	4–46, 4–51, 4–52, 5–1, 5–4, 5–6, 5–8, 5–9, 5–11, 5–12,
	6–11, 6–13, 6–14, 7–11, A–11, A–14, B–13, B–15, B–16,
	B–20, C–1, C–4, C–10, D–3, D–5.
	Most of these pages contain corrections of spelling and language errors.
	Changes of technical significance are marked by a vertical line in the margin.
	Appendix A.3 has been removed. Refer to the manual "SINTRAN III
	User's Guide" for File System Error Messages.

ND–10 BASIC – Compiler Reference Manual
ND-60.071.01

NORSK DATA A.S
P.O. Box 4, Lindeberg gård
Oslo 10, Norway

PREFACE

The product

This manual describes the January 1981 version of the BASIC compiler for ND–100 and NORD–10 computers.

10034B – 32 bit floating point hardware
10024B – 48 bit floating point hardware

The system consists of the software products

2059D – BASIC compiler for 32 bit floating point
2060D – Run time library for 32 bit floating point

or

2000E – BASIC compiler for 48 bit floating point
2001E – Run time library for 48 bit floating point

The reader

This manual is written for anybody who will use the BASIC language for programming and for those who need a user level description of the ND BASIC compiler.

Prerequisite knowledge

No previous experience with either the BASIC language, other programming or computer hardware is expected. A minimum of knowledge of the Sintran III operating system is required in order to log in on the NORD–10/ND–100 system.

The manual

The manual is intended to be read sequentially, and is well suited as a guide to programming in general, using BASIC as a tool. It explains BASIC features and interactive use of the BASIC system in sufficient detail for self study, and contains a complete description of all commands, statements and functions available.

Related documentation:

Sintran III Introduction (ND–60.125)

TABLE OF CONTENTS

+ + +

1.3 *WHAT IS BASIC?*

One such language which is easy to learn and to use is BASIC. BASIC was first developed in 1963/64 at Dartmouth College and has since then been revised several times. An advantage of BASIC is that its rules of form and grammar are quite simple and easy to learn. It is the purpose of this manual to present the language BASIC and to show how it is used to solve simple problems and deal with many situations common in computing. More complicated problems can be solved by combining the simpler steps shown here.

1.4 *WHAT IS ND BASIC?*

1.4.1 *The Language*

ND BASIC is a simple, powerful, high-level programming language that facilitates problemsolving for scientific, business and educational applications run on ND–100 and NORD–10 computers. Among the many programming languages currently in use, the rules and grammar of BASIC must be considered the easiest to learn and use. BASIC permits the user to solve mathematical problems directly from a keyboard printer or an alphanumeric display terminal. BASIC is particularly well suited for timesharing applications since the compiler is re-entrant. This permits multiple users to simultaneously call upon and utilize the same compiler.

The ND BASIC language contains a large number of statement types and functions with special features including matrix operation, alphanumeric information handling, program control and storage facilities and program editing, as well as documentation and debugging aids. Several statements designed expressly to perform matrix computations are incorporated in the operation set. The NORD-10 BASIC has string-, real-, integer-, and double integer variable types. Variable names may consist of up to 7 letters and digits.

1.4.2 *Special Real-Time Facilities*

ND BASIC contains the facilities for linking to external subroutines, including FORTRAN and MAC assembly language libraries, thus making it easy to develop real-time application programs in the BASIC language. This facility makes it possible to use the SINTRAN III real-time capabilities as well as other common processors for control systems.

1.4.3 *Program Development*

ND BASIC provides program control of storage facilities that save programs or data on mass storage devices, and later retrieve them for execution. Program editing permits adding or deleting statement lines from on-line terminals, including possibilities for correcting individual characters of a line, using the same editing facilities as in SINTRAN III command input. Programs may be combined from several source units, requesting a partial or complete hard-copy listing and re-numbering statement lines.

1.4.4 *The Compiler*

The ND BASIC compiler may be used in three different modes:

- Interactive incremental compiler.
- Binary relocatable format (BRF)-compiler.
- Direct execution of statements and expressions.

In the interactive mode lines typed by the user, or read from an existing source file, are compiled into machine-instructions and loaded directly to the user's virtual memory.

When typing the RUN command, the compiled program is executed at highest possible speed, much faster than traditional interpreters. Source lines are kept on a system-scratch-file for later retrieval. Independently compiled subroutines or library files may be linked, using the integrated relocating loader when necessary.

In compile-mode lines are read from existing source files and compiled into binary relocatable format (BRF)-files, compatible with FORTRAN or MAC assembly language subroutines. The BRF file may be loaded for execution by the integrated relocating loader, or by a freestanding loader normally used with FORTRAN/MAC programs.

In immediate mode lines typed without line number are regarded as expressions being compiled into machine instructions, and executed directly. Most statements may be used, with a few exceptions as the FOR/NEXT statements. The terminal may then function as an advanced calculator.

In all modes extensive error messages are given, making it easy to correct erroneous statements.

1.5 *THE MANUAL*

This manual describes the language in steps so that understanding of material presumes a knowledge of material in previous chapters.

1.5.1 *Conventions Used In This Manual*

Some documentation conventions are used in this manual to clarify examples of BASIC syntax. BASIC statements or commands are often described in general terms using the following conventions:

- A statement number is assumed when a statement is described.
- Items in capital letters are reserved BASIC words belonging to the vocabulary of the BASIC language. (RUN, EDIT, IF, LET, STEP.)
- Items in small letters enclosed in < > are essential elements of the statement or command being described. (<statement>, <variable>, <expression>)
- Text enclosed in [ ] is optional.

Some terms which may seem confusing are explained below:

- Terminal is any device having a two-way communication with the computer.
- The user types on the keyboard and BASIC prints on the terminal.
- Capital letters marked with a ^{C} like A^{C} or Q^{C} indicate the respective key on the keyboard plus the CTRL key.

The program and the resulting run is shown below exactly as it appears on the terminal:

```
10  READ A, B, D, E
15  LET G=A*E–B*D
20  IF G=0 THEN 65
30  READ C, F
37  LET X=(C*E–B*F)/G
42  LET Y=(A*F–C*D)/G
55  PRINT X, Y
60  GO TO 30
65  PRINT "NO UNIQUE SOLUTION"
70  DATA 1, 2, 4
80  DATA 2, –7, 5
85  DATA 1, 3, 4, –7
90  END

RUN
 4                 –5.5
 6.66667E–01        1.66667E–01
–3.66667           3.83333

BASIC RUN ERROR 406 IN LINE 30
```

After typing the program, we type RUN followed by a carriage return. Up to this point the computer stores the program and checks the form of the statements. This process is called compiling. It is the RUN command which directs the computer to execute your program. The message out-of-data error code here may be ignored. However, in some cases it indicates an error in the program.

2.2 *EXPRESSIONS*

The computer can perform a great many operations; it can add, subtract, multiply, divide, extract square roots, raise a number to a power and find the sine of a number (on an angle measured in radians), etc.. We will now learn how to tell the computer to perform these various operations and to perform them in the order that we want them done.

The computer performs its primary function (that of computation) by evaluating formulas which are supplied in a program. These expressions are very similar to those used in standard mathematical calculation, with the exception that all BASIC expressions must be written on a single line. Five arithmetic operations can be used to write an expression, and these are listed in the following table.

Symbol	Example	Meaning
+	A + B	Addition (add B to A)
–	A – B	Subtraction (subtract B from A)
*	A * B	Multiplication (multiply B by A)
/	A / B	Division (divide A by B)
↑ or **	X ↑ 2	Raise to the power (find X^2)

We must be careful with parentheses to make sure that we group together those things which we want together. We must also understand the order in which the computer does its work. For example, if we type A + B * C ↑ D, the computer will first raise C to the power D, multiply this result by B and then add A to the resulting product. This is the same convention as is usual for $A + BC^D$. If this is not the order intended, then we must use parentheses to indicate a different order. For example, if it is the product of B and C that we want raised to the power D, we must write A + (B*C)↑D; or, if we want to multiply A + B by C to the power D, we write (A + B)*C↑D. We could even add A to B, multiply their sum by C, and raise the product to the power D by writing ((A + B)*C)↑D. The order of priorities is summarized in the following rules:

- The expression inside parentheses is computed before the parenthesized quantity is used in further computations.
- In the absence of parentheses in an expression involving addition, multiplication and the raising of a number to the power, the computer first raises the number to the power, then performs the multiplication, and the addition comes last. Division has the same priority as multiplication, and subtraction the same as addition.

– In the absence of parentheses in an expression involving operations of the same priority, the operations are performed from left to right.

The rules are illustrated in the previous example. The rules also tell us that the computer faced with A – B – C, will (as usual) subtract B from A and then C from their difference; faced with A/B/C, it will divide A by B and that quotient by C. Given A↑B↑C, the computer will raise the number A to the power B and take the resulting number and raise it to the power C. If there is any question in your mind about the priority, put in more parentheses to eliminate possible ambiguities.

In addition to these five arithmetic operations, the computer can evalute several mathematical functions. These functions are given special English names, for instance:

Functions	Interpretation
ATN (X)	Find the arctangent of X
EXP (X)	Find e^X
SQR (X)	Find the square root of X ($\sqrt{X}$)

In place of X, we may substitute any expression or any number in parenthesis following any of these formulas. For example, we may aks the computer to find $\sqrt{4 + X^3}$ by writing SQR (4 + X↑3), or the arctangent of $3X - 2e^X + 8$ writing ATN (3*X – 2 * EXP (X) + 8).

If sitting at the terminal, you need the value of $(5/6)^{17}$ and you can write the two-line program:

```
10 PRINT (5/6) ↑ 17
20 END
```

and the computer will find the decimal form of this number and print it out in less time than it took to type the program.

Other functions are also available in BASIC, but these are reserved for explanation later (Section B.3).

2.2.1 *Numbers*

A number may be positive or negative and it may contain up to approximately nine significant digits. For example, all of the following are numbers in BASIC:2 –3, 675, 1234567, –7654321 and 483.4156. The following are not numbers in BASIC:14/3 and √7,. We may ask the computer to find the decimal expression 14/3 and √7, and to do something with the resulting number, but we may not include either in a list of DATA. We gain further flexibility by use of the letter E, which stands for "times ten to the power". Thus, we may write 00123456789E – 2 or 123456789E – 11 or 1234.56789E – 6. We may write ten million as 1E7 (or 1E + 7) and 1965 as 1.965E3 (or 1.965E + 3). We do not write E7 as a number, but must write 1E7 to indicate that it is 1 that is multiplied by 10^7.

2.2.2 *Variables*

A variable in BASIC is denoted by any letter, or a letter followed by up to six digits and/or letters. Thus, the computer will interpret E7 as a variable along with A, X, N5, 10 and 01. A variable in BASIC stands for a number, usually one that is not known to the programmer at the time the program was written. Variables are given or assigned values by LET, READ or INPUT statements. The value so assigned will not change until the next time a LET, READ or INPUT statement is encountered with a value for that variable. However, all variables are set to a zero before a RUN command. Thus, it is not necessary to assign a value to a variable before using the variable in a computation.

2.2.3 *Relational Operators*

Seven other mathematical symbols are provided for in BASIC, symbols of relation, and these are used in IF – THEN statements where it is necessary to compare values. An example of the use of these symbols was given in the sample program in Section 2.1.

Any of the following seven relations may be used:

Symbol	Example	Meaning
=	A = B	Is equal to (A is equal to B)
<	A < B	Is less than (A is less than B)
< = or = <	A < = B	Is less than or equal to (A is less than or equal to B)
>	A > B	Is greater than (A is greater than B)
> = or = >	A > = B	Is greater than or equal to (A is greater than or equal to B)
< > or ><	A < > B	Is not equal to (A is not equal to B)
= =	A = = B	Is approximately equal to

The term "approximately equal to" means that the two quantities differ by a very small amount and may be regarded as identical for any practical purpose. More specifically, A == B is true if:

$$|A - B| < C * |(A + B / 2)|$$

C is a system constant which equals 5E–7 for 48 bit reals and 5E–5 for 32 bit reals (see Appendix C).

Generally, approximately equal quantities appear equal when they are printed.

2.3 *LOOPS*

We are frequently interested in writing a program in which one or more portions are performed not just once but a number of times, perhaps with slight changes each time. In order to write the simplest program, the one in which this portion to be repeated is written just once, we use the programming device known as a *loop.*

The programs which use loops can, perhaps, be best illustrated and explained by two programs for the simple task of printing out a table of the first 100 positive integer numbers together with the square root of each. Without a loop, our program would be 101 lines long and read:

```
10 PRINT 1, SQR (1)
20 PRINT 2, SQR (2)
30 PRINT 3, SQR (3)
   .....
990 PRINT 99, SQR (99)
1000 PRINT 100, SQR (100)
1010 END
```

With the following program, using one type of loop, we can obtain the same table with far fewer lines of instruction, 5 instead of 101:

```
10 LET X = 1
20 PRINT X, SQR (X)
30 LET X = X + 1
40 IF X <= 100 THEN 20
50 END
```

Statement 10 gives the value of 1 to X and "initializes" the loop. In line 20 both 1 and its square root are printed. Then, in line 30, X is increased by 1 to 2. Line 40 asks whether X is less than or equal to 100; an affirmative answer directs the computer back to line 20. Here it prints 2 and $\sqrt{2}$, and goes to 30. Again X is increased by 1, this time to 3, and at 40 it goes back to 20. This process is repeated, line 20 (print 3 and $\sqrt{3}$), line 30 (X = 4), line 40 (since $4 \leqslant 100$ go back to line 20), etc. — until the loop has been traversed 100 times. Then after it has printed 100 and its square root, X becomes 101. The computer now receives a negative answer to the question in line 40 (X is greater than 100, not less than or equal to it), does not return to 20, but moves on to line 50, and ends the program. All loops contain four characteristics, initialization (line 10), the body (line 20), modification (line 30), and an exit test (line 40). Because loops are so important and because loops of the type just illustrated arise so often, BASIC provides two statements to specify a loop even more simple. They are FOR and NEXT statements, and their use is illustrated in the program:

```
10 FOR X = 1 TO 100
20 PRINT X, SQR (X)
30 NEXT X
50 END
```

In line 10, X is set equal to 1, and a test is set up, like that of line 40. Line 30 carries out two tasks: X is increased by 1 and the test is carried out to determine whether to go back to 20 or to go on. Thus, lines 10 and 30 take the place of lines 10, 30 and 40 in the previous program — and they are easier to use.

Note that the value of X is increased by 1 each time we go through the loop. If we wanted a different increase, we would specify it by writing:

```
10 FOR X = 1 TO 100 STEP 5
```

and the computer would assign 1 to X on the first time through the loop 6 to X on the second time through, 11 on the third time, and 96 on the last time. Another step of 5 whould take X beyond 100, so the program would proceed to the end after printing 96 and its square root. Step size must be positive, unless it is a negative constant.

In the absence of a STEP clause, a step size of +1 is assumed.

More complicated FOR statements are allowed. The initial value, the final value, and the step size may all be expressions of any complexity. For example, if N and Z have been specified earlier in the program we could write:

```
FOR X = N + 7 * Z TO (Z – N) /3 STEP (N – 4 * Z) /10
```

The loop continues as long as the control variable is algebraically *less than or equal* to the final value.

If the initial value is greater than the final value, then the body of the loop will not be performed at all, but the computer will immediately pass to the statement following the NEXT. For example, the following program for adding up the first n integer numbers will give the correct result 0 when n is 0.

```
10 READ N
20 LET S = 0
30 FOR K = 1 TO N
40 LET S = S + K
50 NEXT K
60 PRINT S
70 GO TO 10
90 DATA 3, 10, 0
99 END
```

It is often useful to have loops within loops. These are called *nested loops* and can be expressed with FOR and NEXT statements. However, they must actually be nested and must not cross, as the following skeleton examples illustrate:

Allowed

```
FOR X
  FOR Y
  NEXT Y
NEXT X
```

Allowed

```
FOR X
  FOR Y
    FOR Z
    NEXT Z
    FOR W
    NEXT W
  NEXT Y
    FOR Z
    NEXT Z
NEXT X
```

Not Allowed

```
  FOR X
FOR Y
  NEXT X
NEXT Y
```

Note that BASIC does not check for overlap of control variables in nested loops.

The last line is always stored in the computer, and you can correct it, even if it resulted in an error message by using the line exit control characters. Any program statement may also be corrected in the same way by typing the EDIT command followed by the statement number. If you want to eliminate the statement on line 110 from your program, you may do this by typing the command DELETE 110. It is also possible to type 110 followed by carriage return. Now, line 110 is still a part of the program, but the effect of the statement is removed. If you want to insert a statement between those on lines 60 and 70, you can do this by giving it a line number between 60 and 70.

If it is obvious to you that you are getting the wrong answers to your problem, even while the computer is running, press the key marked ESC and the control is given to the Operating System. The command CONTINUE will restart BASIC with your program intact and you can start to make your corrections. If you are in serious trouble, type the command CLEAR. The word READY, whenever printed, tells you that BASIC is ready to accept commands or statements from your terminal.

A sample use of the system is shown below:

```
10  FOR N = 1 TO 7
20  PRINT N, SQR(N)
30  NEXT N
50  END

RUN
 1          1
 2          1.41421
 3          1.73205
 4          2
 5          2.23607
 6          2.44949
 7          2.64575
READY
```

2.6 *ERRORS AND DEBUGGING*

It may occasionally happen that the first run of new problem will be free of errors and give the correct answers, but it is much more likely that errors will be present and will have to be corrected. Errors are of two types: errors of form (or syntax errors) which prevent the running of the program, and logical errors in the program which cause the computer to produce wrong answers or no answers at all.

Errors of form will cause error messages to be printed. Logical errors are often much harder to uncover, particularly when the program gives answers which seem to be nearly correct. In either case, after the errors are discovered, they can be corrected by changing lines, by inserting new lines or by deleting lines from the program. As indicated in the last section, a line is changed by typing it correctly with the same line number; a line is inserted by typing it with a line number between those two existing lines; and a line is deleted by typing DELETE and the actual line number. Notice that you can insert a line only if the original line numbers are not consecutive integers. For this reason, most programmers will start out using line numbers that are multiples of five or ten, but that is a matter of choice.

These corrections can be made at any time - whenever you notice them - either before or after a run. Since the computer sorts lines out and arranges them in order, a line may be retyped out of sequence. Simply retype the offending line with its original line number.

As with most problems in computing, we can best illustrate the process of finding the errors (or "bugs") in a program and correcting (or "debugging") it by an example. Let us consider the problem of finding that value of X between 0 and 3 for which the sine of X is a maximum and ask the machine to print out this value of X and the value of its sine. If you have studied trigonometry you know that $\pi/2$ is the correct value; but we shall use the computer to test successive values of X from 0 to 3, first using intervals of .1, then .01, and finally of .001. Thus, we shall ask the computer to find the sine of 0, of .1, .2, .3, ... 2.8, 2.9 and of 3, and to determine which of these 31 values is the largest. It will do it by testing SIN(0) and SIN(.1) to see which is larger and calling the largest of these two numbers M. Then it will pick the larger of M and SIN (.2) and call it M. This number will be checked against SIN (.3) and so on down the line. Each time a larger value of M is found, the value of X is "remembered" in X0. When it finishes, M will have been assigned to the largest value. It will then repeat the search, this time checking the 301 numbers 0, .01, .02, .03, ... 2.98, 2.99, and 3, finding the sine of each and checking to see which has the largest sine. At the end of each of these three searches, we want the computer to print three numbers: the value X0 which has the largest sine, the sine of that number, and the interval of search.

3 INTERACTIVE USE OF THE BASIC SYSTEM

3.1 *ENTERING THE BASIC SYSTEM*

The BASIC system may be entered from the operating system SINTRAN III by typing

```
@BASIC
```

Then the BASIC system starts by identifying itself followed by the word READY. This word, whenever printed, tells you that BASIC is ready to accept a command or a statement typed from your terminal.

3.1.1 *Compiling a BASIC Program*

When you start, the system is initialized to accept your program lines typed directly from the terminal. However, if your program resides on a mass storage file you may initiate the compilation process by giving the command:

```
OLD <file name>
```

As soon as all the program lines on the specified file has been compiled, the number of compiled lines along with the number of diagnostics given will be printed on your terminal. If no diagnostics are given the compiler has accepted all the statements to be syntactically and semantically correct and you may try to start the execution of it (see below).

3.1.2 *Editing a BASIC Program*

If compiler diagnostics have occurred these must be corrected before the program can be executed. The BASIC system provides commands to list, delete, change and renumber the program lines.

A line may be changed simply by typing a new line with identical line number. Then the new line will replace the old one.

A line may also be changed by first typing

```
EDIT <line number>
```

and then applying the line edit control characters to produce a modified line. The control characters are described in Appendix C.3.

Example:

```
10 LET A =,1
***ERROR IN LINE 10", "SYNTAX ERROR"
ED 10
```

Now if Z^C followed by = is typed this will result in the printout:

```
10 LET A =
```

Then if a S^C is typed in order to remove the comma, D^C will copy the rest of the old line to the new one.

A line may be listed on terminal by typing

LIST <line number>

Now this line may be modified without using the EDIT command. More than one line may be specified, each line number separated by comma. The word LIST by itself will cause the listing of the entire program.

LIST followed by two line numbers separated by a dash (-) will list the lines between and including the specified ones.

A line is removed from the program by typing

DELETE <line number>

More than one line may be specified, separated by commas. Two line numbers separated by a dash (-) will delete the lines between and including the lines specified.

The RENUMBER command is used to change statement line numbers and the references to these lines. Line numbers in comments are not changed.

A program is renumbered by typing

RENUMBER <new initial line number> [<increment>]

First parameter indicates the new initial line number and the second (if any) indicates the increment in the line numbers of two successive statements. If no parameters are specified the first statement number will be 100 and the increment will be 10.

(A + B)	(–A * B)	((A**B)–(A*B))
124	12.4E–2	0%
X	A(I, J)	SIN(V)

A factor is a primary or a primary ** a primary:

(A + B)	(A + B)**X	I**2

A term is a factor, a term/factor, or a term*term:

A**B	(A**B)/X	((A**B)/X)*SIN(V)

A signed term is immediately preceded by a plus or minus:

–A**B	–X	–(–A*B)

A simple arithmetic expression is a term, or two simple arithmetic expressions separated by plus or minus:

(A + B)+X%	X/2.314	Y/SIN(X)–A**B

An arithmetic expression is a simple arithmetic expression, or a signed term plus or minus a simple arithmetic expression:

–X/Y	I**2 + K%	–A**B–X/Y

4.2.3 *Rules for Forming Expressions*

Two arithmetic operators may not be adjacent to each other, X + –Y is an illegal expression. The subtraction operator may not be used as a sign of negation. –X implies 0–X and must be enclosed in parentheses when preceded by another operator: X + (–Y) is a legal expression.

Parentheses may be used to indicate grouping as in ordinary mathematical notation but they may not be used to indicate multiplication: (X) (Y) does not imply (X) * (Y) nor does juxtaposition imply multiplication: XY does not imply X * Y. Real and integer quantities may be mixed in the same expression.

4.2.4 *Order of Evaluation*

When the hierarchy of operations in an expression is not completely specified by parentheses, the operations are performed in the following order:

↑ or **	exponentiation	performed first
/	division	performed next
*	multiplication	
+	addition	performed last
–	subtraction	

Within a sequence of consecutive multiplications and/or divisions or additions and/or subtractions, when the order is not explicitly indicated by parentheses, expressions are evaluated from left to right.

Whenever ambiguity is possible in the evaluation of an expression, parentheses should be used. The ambiguous expression A**B**C can be clarified as (A**B)**C or A**(B**C) only by parentheses.

The way an expression is written determines how the computer will evaluate it.

1. 10 ↑ 2+1

   The computer evaluates this expression as 100 + 1 = 101. It will perform the exponentiation before the addition.

2. 10 ↑ 2/2*3

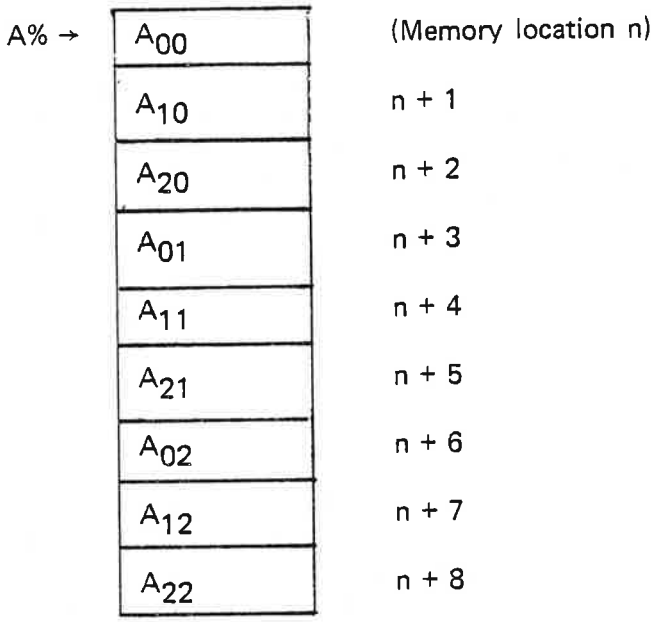
   The value given for this expression is 100 / 2 * 3 = 50 * 3 = 150. The computer performs the exponentiation first. When multiplication and division appear together, the left-most operation is performed first. Thus, in this example, the division is performed second and finally the multiplication.

3. 5 + 2 * 3 – 1

   The value of this expression is 5 + 6 – 1 = 11 – 1 = 10. The computer performs the multiplication first. As with multiplication and division, the positions of the + and – symbols determine which operation is performed first. Addition and subtraction are performed from left to right. So, in this example, the addition is performed second and the subtraction last.

4. 32 / 4 ↑ 2 + 3 * 3 – 1

   This expression uses all the available symbols for arithmetic operations and the steps by which the computer evaluates it are as follows. First exponentiation is performed and the expression is reduced to 32 / 16 + 3 * 3 – 1. Then division and multiplication are performed from left to right and the simplified formula is 2 + 9 – 1. Finally, addition and subtraction are performed from left to right and the value of the formula is seen to be 10.



The location of an array element with respect to the first element is a function of the maximum array dimensions and the type of the array. Given DIM A% (L, M), the location of A% (I, J) with respect to the first element of array A% is given by:

$$A\% + [I + (J * (L + 1))] * E$$

The quantity in brackets is the subscript expression. E is the element length in terms of the number of computer words needed for each element of the array. In our example, where the array (A%) is of integer type E is equal to 1. For string arrays E will always be equal to 2, because such arrays, in fact, consist of pointers to the string elements, and the length of each.

4.5 *FUNCTIONS*

With the BASIC statements previously described, programs can be written which compute values of many of the commonly used elementary functions. For example, the following portion of a BASIC program can be used to find the absolute value of a number N and store it in A.

```
220     REM SIGNED NUMBER IN N
230     IF N < 0 THEN 260
240     LET A = N
250     GO TO 270
260     LET A = (-N)
270     REM POSITIVE NUMBER IN A
```

Because the need for the absolute value of a number arises so frequently in programming, BASIC provides a simpler way of computing this function. Certain elementary function names (such as ABS) may appear in BASIC programs anywhere a number may appear. The function name is followed by any arithmetic expression enclosed in parenthesis. For example, the absolute value of a number may alternatively be calculated with the following portion of a BASIC program:

```
220     REM SIGNED NUMBER IN N
230     LET A = ABS (N)
240     REM POSITIVE NUMBER IN A
```

BASIC computes the value of these functions accurately, it does not store tables of elementary functions, since it can compute a value for a function in a few thousandths of a second. If a number which cannot be evaluated is used with a function, a message is printed on the terminal. For example, if a program attempts to take the square root of a negative number.

Most of the function names are self-explanatory. The range of the arctangent function ATN is from −pi/2 to +pi/2. The function INT(X) delivers the largest integer number not greater than X, for example:

```
INT (-2.8) = -3
INT (2.8) = 2
INT (-.0001) = -1
```

The INT function can be used to good advantage to round numbers:

```
100 LET A = INT (A + .5)
110 LET B = INT (100 * B + .5)/100
```

Statement 100 rounds A to the nearest integer. Line 110 rounds B to the nearest hundreth.

Function calls may be nested. The following program prints the sine of the angle whose arctangent is T.

```
10 INPUT T
20 PRINT SIN (TAN(T))
30 END
```

4.5.1 *Function Classification*

Functions in ND BASIC are divided into three main groups:

1. Mathematical functions
2. String functions
3. Miscellaneous functions

These three types of functions can be defined for a BASIC program in several ways:

1. Built-in library functions

   Functions with restricted names; most commonly used in programs.

2. Extended library functions:

   Existing functions which may be supplied by scanning a library file.

3. User internal functions:

   Any desirable function defined by the user through a DEF statement. The name must start with FN.

4. User external functions:

   Any desirable function introduced in a BASIC program through an EXTERNAL statement. The function must be present in the NORD standard object form (BRF); the source code, however, may be BASIC, STANDARD FORTRAN, NPL or MAC assembly.

When a function reference appears in a BASIC program, the compiler generates a calling sequence within the object program.

All existing functions are listed with a short description in Appendix B.3. The way of defining and calling user functions are described later.

4.6 *REPRESENTATIONS OF STRINGS*

The BASIC programs described thus far have all dealt with numbers. In the statement

```
100 LET A = B + 3.1415926
```

the sequence 3.1415926 is a representation of a number; the character B is the name of a number which can vary as the program is executed by the computer. The character A is the name of a number which may be changed by the execution of that statement. Although computers are excellent machines for performing high-speed arithmetic, some of their most important uses are in the manipulation of entities which *do not* represent numbers. A string is such an entity.

A *string* is a sequence of characters; these include letters, digits, blanks, and other special characters such as those which appear on the terminal. One way of representing a string in BASIC is to enclose it in quotation marks. Such string constants have already been introduced in INPUT and PRINT statements. For example, the string in

```
100 PRINT "NO UNIQUE SOLUTION"
```

is a string constant just as the number 3.1415926 in the preceding example is a numeric constant.

Just as BASIC has names for numbers, it also has names for strings. A name of a simple string is formed exactly as a name for a number, except that it includes a trailing dollar sign ($). The string A$ is entirely distinct from the number A and both names can appear in the same BASIC program.

4.6.1 *Assigning Values to Strings and String Comparisons*

A string variable can take on a string value through a READ statement. The following BASIC program reads three strings and prints them.

```
10 READ A$, B$, C$
20 PRINT C$; B$; A$
30 DATA "ING", "SHAR", "TIME "
40 END
```

Note that the items in the DATA statement are representations of strings, not numbers. This program prints the word TIMESHARING on the terminal. Since the quotation marks are used to delimit the strings, it is not possible to create a string containing a quotation mark in this manner.

Strings can also be assigned values through the use of LET statements. For example:

```
10 LET A$ = "H2SO4"
20 LET B$ = A$
30 PRINT B$
40 END
```

will print the string H2SO4 on the terminal. It is even possible to omit the word LET as with arithmetic assignment statements.

Another way that a string can take on a value is by having the program request the input of a string from the terminal through an INPUT statement. For example:

```
10 PRINT "A MIXTURE OF FUEL AND OXIDIZER WHICH"
20 PRINT "BURNS SPONTANEOUSLY IS TERMED";
30 INPUT A$
40 IF A$ = "HYPERGOLIC" THEN 70
50 PRINT "WRONG"
60 GO TO 80
70 PRINT "RIGHT"
80 END
```

After printing the textual message the program will print a question mark. Suppose the user enters the word "HYPERVENTILATED" in response. Statement 40 is a string conditional statement. If the string A$ is the same as the string "HYPERGOLIC", then statement 70 will be executed next. Since the user did not enter "HYPERGOLIC" he has WRONG printed on his terminal.

Any of the relational operators except approximately equal (described in Section 2.2.3) may be used in an IF – THEN statement to compare strings. The relational operator "<" is interpreted as meaning "earlier in alphabetical order than" and the relational operators are defined appropriately. The ordering of characters is arbitrarily defined by the ASCII code which is explained in Appendix C.4. In any string comparison the strings are assumed to be of the same length, i.e., trailing blanks are simulated.

4.6.2 *Relaxation of Requirement for Quotation Marks*

Strings which are entered in response to an INPUT statement need not be bracketed by quotation marks as long as the items being entered do not contain commas or do not begin with blanks.

Strings containing commas must be enclosed in quotation marks because commas are treated as special characters by BASIC. They are used to separate multiple items entered in response to an INPUT statement containing more than one variable in the input list. In addition, if the last string on a line of input being entered in a list via a MAT INPUT statement ends with an ampersand (&), the string must be enclosed in quotation marks.

A string in a DATA statement must be enclosed in quotation marks if it begins with a blank, a digit, a plus sign, a minus sign, or a decimal point, or if it contains a comma or an apostrophe. Ampersands, however, do not have the special significance in DATA statements that they do in items being entered in response to INPUT statements. If strings are enclosed in quotation marks, the quotation marks are not considered to be part of the string and are ignored.

4.6.3 *More About RESET*

In DATA statements, numbers and strings may be intermixed. When a numeric variable appears in a READ statement the next number appearing in the DATA statements is assigned to that numeric variable; when a string appears in a READ statement, the next string appearing in DATA statements is assigned to that string variable. Thus, numeric and string data are managed independently in BASIC. A RESET statement will reset pointers for both types of data so that subsequent READ statements will reread the data. A RESET * statement will reset only the pointer for string data.

The following program illustrates the use of RESET.

```
100 READ A$, A, B$
110 PRINT "FIRST TIME", A$, A, B$
120 DATA 1, "2APPLES", PEARS
130 RESET
140 READ C$
150 PRINT "SECOND TIME", C$
160 END
```

Running this program produces the following input:

```
FIRST TIME     2 APPLES     1     PEARS
SECOND TIME    2 APPLES
```

4.6.4 *String Arrays*

BASIC can also operate on multiple strings arranged as one or two dimensional arrays. These entities are denoted by a string identifier, followed by one or two subscripts enclosed in parenthesis. Thus A$(3) denotes the third string in a list of string A$. Similarly, B$(4, 5) denotes a string in the 4th row and 5th column of a table of strings B$.

The SEG$ Function

SEG$ (A$, X, Y) takes a string and two expressions as arguments and returns a substring as a result. The substring starts at character number X in the input string and ends at character number Y.

Example:

```
50 LET NEW$ = SEG$ (A$, 3, 3) & B$
```

4.7 *FORMATTING OUTPUT*

When you write BASIC programs to prepare reports, graphs, tables and other formatted (or specially arranged) output, it is important that you will be able to control output format very closely. This section describes statements which permit construction of neatly aligned tables, labels and so on.

4.7.1 *Exclamation Marks in PRINT Lists*

The exclamation mark (!) will cause the terminal print head to move to the next line, i.e., carriage return and line feed is printed. This will be repeated for each exclamation mark found as in the example:

```
10 PRINT !, 1, !!, 2
20 END
RUN

 1

 2
```

4.7.2 *Commas in PRINT Lists*

The terminal line is considered to be divided into zones of 15 characters each. The default number of zones is 5 as the standard margin (see Section 4.7.7) is set to 75. Each line begins with column zero.When multiple items appear in a PRINT list separated by commas, the first item is printed starting at the beginning of the first zone (column 0), the second at the next zone (column 15), etc. The comma can be considered to cause the terminal print head to space up the next zone preparatory to printing. If the last zone has just been filled, the terminal print head will move to the first print zone of the next line. Thus, the statement

```
100 PRINT , , , , "COL60"
```

will print the five character "COL60" beginning at column 60, the beginning of the fifth zone.

If a PRINT list ends in a comma, the terminal print head simple spaces up to the next 15 character zone and does not move to the beginning of a new line in preparation for the next PRINT statement unless the last zone has been filled.

For example, the program:

```
100 FOR I = 1 TO 15
110 PRINT I,
120 NEXT I
130 END
```

```
 1     2     3     4     5
 6     7     8     9     10
 11    12    13    14    15

READY
```

4.7.3 *Empty PRINT Statements*

A PRINT statement which does not end in any special punctuation mark, such as a comma, will print the information in the PRINT list and the terminal will be prepared so that further output will begin at the beginning of the next line. Thus, an empty PRINT statement such as

```
100 PRINT
```

will simple advance the paper one line, leaving a blank line if the terminal print head is already at the beginning of a line. It can be used to cause the completion of a partially filled line as illustrated in the following program.

```
100 FOR I  = 1 TO 4
110 FOR J = 1 TO I
120 LET B(I, J) = I
130 PRINT B (I, J),
140 NEXT J
150 PRINT
160 NEXT I
170 END
```

This program will print B(1,1) on the first line. Without line 150, the terminal print head would then go on printing B (2, 1), B (2,2) on the same line. Line 150 directs the terminal print head to start at the beginning of a new line after printing the highest J value for a given I. Thus, items are printed in a triangular format. Output from the preceding program follows:

```
1
2     2
3     3     3
4     4     4     4

READY
```

4.7.4 *Packed PRINT Lists*

Using the comma to separate items in PRINT lists, you will find that it is not possible to print more than five numbers or strings on one line. A semicolon may be used to print items closely packed on a line. For example, the program

```
100 FOR I = 1 TO 15
110 PRINT I;
120 NEXT I
130 END
```

will cause the following output to be printed.

```
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15

READY
```

To determine what will be printed using the semicolon separator, it is necessary to know how strings and numbers are printed. In general, when you use the semicolon to format output, no blanks will be output other than those automatically output when a number is printed as described in the following section.

4.7.5 *Printing Formats for Numbers and Strings*

This section describes the spacing of numbers and strings as they are printed by a simple PRINT statement.

Strings are printed just as they are, with no leading or trailing spaces. A space is printed after the right-most digit of a number; negative numbers are preceded by a minus sign and positive numbers are preceded by a blank.

The number of spaces which will be occupied by the decimal representation of a number varies according to the magnitude and type (integer or non-integer) of the number. The following discussion of how numbers are printed will help in determining the expected printed output.

Numbers may be printed using one of three notations:

I A number printed using *integer* notation is printed without a decimal point and contains from 1 to 6 digits. (For example, twenty printed as 20 is in integer notation.)

II A number printed in *fractional* notation contains from 1 to 6 digits and a decimal point. Trailing (right-most) zeros are dropped and a number less than one is printed with a zero to the left of the decimal point. (For example, twenty printed as 20. is in fractional notation.)

DEF statements may involve both dummy arguments and variables which have the same meaning as elsewhere in the program. In the following example

```
100 DEF FNX (X, Y) = X * COS(T) + Y * SIN(T)
110 DEF FNY (X, Y) = ÷ X * SIN (T) + Y * COS(T)
120 LET T = 1.7 'ANGLE IN RADIANS
130 INPUT A, B
140 PRINT "ROTATED", FNX(A, B), FNY(A, B)
150 GO TO 130
160 END
```

the DEF statements involve both the dummy variables X and Y whose values depend on the arguments of the function and a variable T which has the same value as it does elsewhere in the BASIC program. If a variable in a DEF statement is to have its current value in the program when the function is called, it is not included in the list of dummy arguments. It is often called a *global* variable.

4.11.2 *Multiple Line DEF Statements*

The use of the DEF statement described above is limited to those functions which can be defined in a single BASIC arithmetic statement. Many functions cannot be computed using a single BASIC arithmetic expression, particularly those which require IF — THEN statements. The following example demonstrates the format of multiple line DEF statements and their use for a function which returns the larger of two numbers.

```
10 DEF FNM (X, Y)
20 LET FNM = X
30 IF Y <= X THEN 50
40 LET FNM = Y
50 FNEND
55 '
60 PRINT FNM (5,4), FNM (–5, –4)
70 PRINT FNM(1, FNM(2, FNM(3,0)))
80 END
```

The definition of the function extends from line 10 to line 50.

The absence of the equal sign in line 10 indicates that this is a multiple line DEF; the end of the DEF is indicated by the FNEND statement. The value which the function delivers must be stored in the variable having the same name as the function (in this case, FNM) when control reaches the FNEND statement. As illustrated in line 70, function calls may be nested. The preceding program prints the numbers 5, –4, and 3.

As with the single line function definition, variables appearing in parentheses after the function name in a multiple line definition are called dummy arguments, and values are substituted for these arguments when the function is called. Variables not listed in the DEF statement will use their current value. There must not be a transfer from inside a multiple line DEF to outside, nor vice versa. Function definitions may not be nested. Naming conventions are the same as for single line definitions. Multiple line function definitions may be placed anywhere in a program because such blocks of code are not executed, unless they are called.

If a value is not stored as in line 40 above, for the function when control reaches the FNEND statement, a value of zero is returned when the function is called. Any variable assignments made to variables other than the dummy arguments of the function within the scope of a multiple line definition affect the values of variables of the same name appearing elsewhere in the program.

4.11.3 *Strings and Function Definitions*

The function definitions described thus far delivered numbers as results and take numbers as arguments. A function may be defined which takes strings as arguments.

Example:

```
100 DEF FNN (A$, B$) = ABS(LEN(A$) – LEN(B$))
110 INPUT Q1$, Q2$
120 PRINT "STRING LENGTHS DIFFER BY"; FNN(Q1$, Q2$)
130 GO TO 110
140 END
```

The following function inserts string B$ after the n'th letter of string A$ and delivers a string as the value of FNI$.

```
100 DEF FNI$ (A$, B$, N)
110 LET C1$ = SEG$ (A$, 1, N)
120 LET C2$ = SEG$ (A$, N + 1, LEN(A$))
130 LET FNI$ = C1$ &B$ & C2$
140 FNEND
150 '
160 PRINT FNI$ ("XXXZZZ", "YYY", 3)
170 END
```

When run, this program prints the string "XXXYYYZZZ".

4.14.4 *The ON ERROR GOTO Statement and the ERR Variable*

ON ERROR GOTO <line number>

In Appendix A, a complete list of run-time error messages is given. The occurrance of errors marked FATAL will normally cause termination of program execution, while non-fatal errors will continue after some action has been taken. A negative argument to the square root function, for example, results in printing a message and continuing with the result set to zero. However, an input/output error such as encountering end of file is fatal.

Some applications may require continued execution of a program after any errors occur. In these situations, you can execute an ON ERROR GOTO statement within your program. This statement tells BASIC that a user subroutine exists, beginning at the specified line number which will analyze any error encountered in the program and possibly attempt to recover from the error. Note that the GOTO action is *not* taken when executing ON ERROR GOTO, but if an error occurs later on, execution is interrupted and the user written subroutine is started at the line number indicated without printing any message. ON ERROR GOTO must be executed prior to any executable statement with which the error handling routine deals.

A system variable, ERR, is available and can be tested according to the error codes given in Appendix A. Thus, the error handling routine can determine precisely what error occurred and decide what action is to be taken. It is possible to switch to different error handling routines by executing several ON ERROR GOTOs.

Often, it is desirable to let the system handle errors in portions of a program. The actual error routine can be disabled by executing ON ERROR GOTO 0. The occurrence of zero, which cannot be a line number, causes the system to treat errors as if ON ERROR GOTO had never been executed.

Example:

```
10 PRINT 1/0
20 ON ERROR GOTO 100
30 PRINT 1/0
40 STOP
100 PRINT "DECIMAL ERROR CODE=";ERR
110 PRINT "OCTAL ERROR CODE=";OC$(ERR)
120 ON ERROR GOTO 0
130 PRINT 1/0
200 END
RUN

BASIC RUN ERROR      273 IN LINE 10
0
DECIMAL ERROR CODE = 187
OCTAL ERROR CODE = 00000000273

BASIC RUN ERROR      273 IN LINE 120
0

READY
```

4.14.5 *The @ Statement*

@ <operating system command>

This statement provides a means to execute SINTRAN III Commands in the program sequence or in immediate mode. The command may be of any type, such as deleting a file, reading the clock or even logging out! Note that error conditions will return control to the Operating System. (Restart with CONTINUE.)

Example:

```
10 @TIME-USED
20 REPEAT 50000: N = N + @
30 @TIME-USED
40 PRINT !,N,!
50 @LOG
60 END
RUN

TIME USED IS     1 SECS OUT OF          41 SECS
TIME USED IS     5 SECS OUT OF          48 SECS
1.25002E+09
15.13.58         26 APRIL     1976
--EXIT--
```

4.14.6 *RANDOM and RND*

The RANDOM statement can be used in conjunction with the random number function to induce variance. It augments the function RND by causing it to produce different sets of random numbers. For example, if this is the first instruction in the program using random numbers, then repeated program execution will generally produce different results. When this instruction is omitted, the "standard list" of random numbers is obtained.

It is suggested that a simulation model should be debugged without RANDOM, so that you always obtain the same random numbers for test runs. After your program is debugged, you may insert

```
1 RANDOM
```

before execution.

5 FILES IN BASIC

5.1 *INTRODUCTION*

Files are the retrievable units in which information is stored. All the programs discussed so far in this manual are examples of files. Files are classified according to how the information is accessed.

Sequential files are accessed one character after the other. In Chapter 3, the saving and retrieval of *program* files are explained. These files are sequential files.

Data in random access files are accessed using an address. If data is used in random manner, retrieval using an address is normally much faster than sequential searching. In BASIC random files are used to hold data arrays too big for the memory available but still manipulated using BASIC programs.

BASIC utilizes the NORD File system through a set of different monitor

The File System is designed to manipulate files on disks, drums, magnetic tapes, cassette tapes or standard peripherals. A file means a collection of records or blocks, ordered randomly or sequentially.

Each file in the system is named with a character string and has one owner, which has to be defined as a user of the file system. Each user may have several other users as friends. The file system provides individual protection of files. with separate protection modes for the owner, the owner's friends and the public's access of the file.

The user of the file system may treat files on mass storage devices or standard peripherals in a uniform manner.

The NORD File System is described in detail in the documentation:

SINTRAN III Timesharing/Batch Guide (ND—60.132)
SINTRAN III Reference Manual (ND—60.128)

5.1.1 *The Connect Device Identifier*

When accessing a file through any BASIC input/output statement, a so-called connect device identifier is used, rather than the file name. The file name is only referenced once, in the OPEN statement which is described below. It is also possible to access a sequential file if the file is opened by a direct file system command. In this case, the connect device identifier must correspond to the file system logical device number. Later we shall see that the connect device identifier may be a string, thus simulating sequential input/output devices.

The connect device identifier may follow any legal statement having connection with input/output operations and has the general form:

<expression> :

The colon delimiter may be exchanged with the comma delimiter in input/output statements (INPUT, PRINT, etc.).

5.1.2 *The OPEN and CLOSE Statements*

The OPEN statement is used both to associate a number with a file in the file system and to describe how the file should be used. Such a description is valid until the CLOSE statement is used or the file is closed by the system.

OPEN

OPEN # <expression> : FOR <access mode> <file name>

The first expression is the connect device and may be any numeric expression. The access mode must be one of the words listed below:

INPUT	Sequential read access
OUTPUT	Sequential write access
APPEND	Sequential write append
RANDOM	Random read/write access

The file name may be any string expression. The OPEN statement assigns a file to a number, thereafter all references to the file are made through the number. There may be up to 10 open files with a program. The connect numbers may be of any range and need not be assigned sequentially. The open statement must, of course, be executed before any access to the file is made.

A successful OPEN statement demands an entry in the file table where connect number and access information is stored.

CLOSE

CLOSE # <expression> :

The expression indicates the connect number and has the same value as the expression in the OPEN statement.

The CLOSE statement is used when you are finished using a file. The statement will set the file ready to be opened again and leave an empty entry in the file table.

All files should be closed before the end of program execution. This is very important when using random access files because the CLOSE statement causes output of the last block.

Examples:

```
 10 INPUT "FILENUMBER", UNIT, "FILENAME", UNIT$
 20 OPEN # UNIT: FOR INPUT UNIT$
  :
  :
100 PRINT # UNIT, A, B, C, D, E
  :
  :
  :
190 CLOSE # UNIT :
200 END
```

5.2 *SEQUENTIAL FILES*

In this chapter, storing and loading of data on files is discussed. The ways of entering data into a program using the READ and DATA statements or the user terminal (INPUT statement) are both inefficient when the amount of data increases beyond a few items.

Using files, there is almost no limit to the number of items the program can process in one run. There are limits on the length of a program to be compiled and these limits include the DATA statements. Another advantage is that since the program file is never modified (as it would have to be if DATA statements were used), there is no chance of the program itself being inadvertently changed during the typing of a new data set.

5.2.1 *Reading a Sequential File from a Program*

Throughout the next few sections of this chapter, several versions of the same fundamental program will illustrate the use of the statements related to sequential files. This program computes an average grade for each of several students in a group.

The first version of this program, AVERAGE1, uses data stored in a sequential file called GRADES.

A listing of AVERAGE1 follows:

```
100 REM PROGRAM NAME – AVERAGE1
110 '
120 REM THIS PROGRAM COMPUTES AVERAGE GRADES FOR
130 REM A SET OF STUDENTS. EACH STUDENT IS ASSUMED
140 REM TO HAVE THE SAME NUMBER OF INDIVIDUAL
150 REM GRADES TO BE AVERAGED. THE DATA IS IN A
160 REM SEQUENTIAL FILE CALLED "GRADES".
170 REM THE FIRST LINE CONTAINS S, THE NUMBER OF
180 REM STUDENTS, AND G, THE NUMBER OF GRADES PER
190 REM STUDENT. THE REST OF THE FILE CONSISTS OF
200 REM S SETS OF (G + 1) LINES. THE FIRST LINE IN A SET
210 REM CONTAINS THE NAME OF A STUDENT, AND THE
220 REM FOLLOWING G LINES IN THE SET EACH CONTAIN
230 REM ONE OF THE STUDENT'S GRADES.
240 '
250 OPEN # 1: FOR INPUT "GRADES"
260 PRINT "NAME", "AVERAGE"
270 PRINT
280 INPUT # 1 : S,G
290 FOR I = 1 TO S
```

```
300 LET A = 0
310 INPUT # 1 : N$
320 FOR J = 1 TO G
330 INPUT # 1 : X
340 LET A = A + X
350 NEXT J
360 LET A = A/G
370 PRINT N$,A
380 NEXT I
390 CLOSE # 1 :
400 END
```

In AVERAGE1 only one file, GRADES, is used. The OPEN # statement assigning the file GRADES to file number 1 is in line 250. Thereafter, the file GRADES is referred to as file # 1 in lines 280, 310, 330, and 390 of the program.

The INPUT # statement differs from the simple INPUT statement only by the inclusion of the number sign, a file number and a colon. Any list of variables that is legitimate in a simple INPUT statement is also legitimate in an INPUT # statement. See Section 2.7.13.

Now, let us briefly run through the whole program before going on to consider the construction of the data file GRADES. Lines 100 - 230 are remarks describing the program, its limitations and instructions for using it. The OPEN statement has already been described. Lines 260 and 270 print a heading for the output. Line 280 requests the input of two numbers, S and G, from file # 1, the file GRADES. S is the number of students and G is the number of grades per student. A loop indexed by I begins in line 290 and continues through line 380. The program ends after this loop has been executed S times, once for each individual whose grades are to be averaged.

Within this loop, line 300 initializes A, the variable used to store the sum of the grades for an individual. Line 310 requests the input of a string from file # 1, GRADES. This string is the name of the next individual whose grades are to be averaged. Another loop begins in line 320 and ends in 350. This loop is executed G times, once for each grade. Within the loop indexed by J, line 330 inputs a grade, X, from GRADES and line 340 adds this grade to A, the sum of the grades so far. When this loop has been executed G times, line 360 divides the sum of the grades, A, by the number of grades, G, to get the average grade which is stored in A. Line 370 prints the name of the individual, N$, and his average, A. Then the loop indexed by I is executed for the next individual, until all averages have been computed and printed.

Now let us consider the data file. The format used in constructing a sequential file to be read by a program is determined by the way in which the INPUT # statements are set up in the program. INPUT # statements, like simple INPUT statements, contain lists of variables to receive values. Whereas a simple INPUT statement requests the user of the program to supply these values at run time, the INPUT # statement requests the values from files, and, of course, no question mark is printed on the terminal. It considers the contents of the next line in the file (beginning with the first line in the file), as a response to its request. If there are more numbers or strings in the line than were requested, the excess is ignored. If there are not, the next line in the file is interrogated in an attempt to find more numbers or strings. If the items on the line interrogated do not correspond in type to the variables in the input list, an error message is printed.

The first INPUT # statement in AVERAGE1 requests two numbers, S and G. These numbers may either be on the same line in the data file or on two different lines. The rest of the numbers and strings in GRADES must be written one per line since they will be read by INPUT # statements requesting one number at a time. If they were erroneously written more than one per line, all but the first number on each line would be ignored, the computer would look for values beyond the end of the file and the program run would terminate. The file GRADES must not have line numbers – just the data requested by the INPUT # statements in the program. The following is a listing of the file GRADES as written for use with AVERAGE1. Note that when more than one item is listed on the same line, the items are separated by commas, as in the first line of GRADES.

```
3,4
GERALD FRIEND
78
86
61
90
PHILIP CLOUGH
66
87
88
91
ADA SHAW
56
77
81
85
```

This file could be created by using the PED editor.

(For information about PED consult the PED User's Guide (ND–60.124)).

The following is a run of AVERAGE1 using the data in the file GRADES:

```
AVERAGE1
NAME              AVERAGE

GERALD FRIEND 78.75
PHILIP CLOUGH  83
ADA SHAW       74.75

READY
```

5.2.2 *Writing a Sequential File from a Program*

In this section, we will consider how to alter the program AVERAGE1 so that it writes its output into a sequential file instead of printing it on the terminal. Using a file in this manner allows the user to obtain multiple copies of the output without re-running the program. In addition, if there is a lot of output, it is often more convenient and possibly faster to direct the output to a file and then list the file than to print the output directly on the terminal.

Two changes need to be made in AVERAGE1; first, another OPEN statement must be added to assign the output file to a file number; and second, the simple PRINT statements must be changed to PRINT # statements. The following program, AVERAGE2, incorporates these changes. The output is printed in a sequential file called AVERAGES.

```
210 REM PROGRAM NAME – AVERAGE2
220 '
230 REM THIS PROGRAM IS LIKE AVERAGE1 EXCEPT THAT
240 REM THE OUTPUT IS PRINTED IN A SEQUENTIAL
250 REM FILE CALLED "AVERAGES".
270 '
290 OPEN # 1: FOR INPUT "GRADES"
300 OPEN # 2: FOR OUTPUT "AVERAGES"
310 PRINT # 2: "NAME", "AVERAGE"
320 PRINT # 2:
330 INPUT # 1:S,G
340 FOR I = 1 TO S
350 LET A = 0
360 INPUT # 1:N$
370 FOR J = 1 TO G
380 INPUT # 1:X
390 LET A = A + X
400 NEXT J
410 LET A = A/G
420 PRINT # 2:N$,A
430 NEXT I
440 CLOSE # 1:
450 CLOSE # 2:
460 END
```

The input file GRADES is assigned to file # 1 and the output file AVERAGES is assigned to file # 2.

When the program is run, line 300 will set the file AVERAGES ready to receive output. Any information in the file will be destroyed and you should do as follows if you want to save the information:

1. Enter the editor PED (see above)

2. Read the file

3. Save the file using a new name

It is still easier to use the SINTRAN III Operating System command: COPY.

After the program AVERAGE 2 has been run, you can list the file AVERAGES using COPY or the PED editor. The following printout results:

```
NAME              AVERAGE

GERALD FRIEND     78.75
PHILIP CLOUGH     83
ADA SHAW          74.75
```

Note that the output of AVERAGE2 and that of AVERAGE1 is identical; the only programming difference is that the first program prints its output to a file and AVERAGE1 prints output directly on the terminal. The format of the output in AVERAGES is the same as that of the output printed on the terminal when AVERAGE1 is run.

5.2.3 *The Use of the Terminal Itself as a File*

Suppose now that we wanted to rewrite AVERAGE2 so that the use of files for input and output was optional. We could write separate sections in the program to deal with each option and then to branch to the appropriate section. However, there is an easier way. Both the INPUT # and the PRINT # statements interpret a reference to file number 0 as a reference to the terminal itself and in this case work exactly like the simple INPUT and PRINT statements.

The following program, AVERAGE3, is a revision of AVERAGE2 in which the user may decide whether or not he wishes to use files. In addition he may choose the names of the data and output files if he wants to use files.

```
100 REM PROGRAM NAME – AVERAGE3
110 '
120 REM THIS PROGRAM IS LIKE AVERAGE2 EXCEPT
130 REM THAT THERE ARE OPTIONS FOR READING
140 REM DATA FROM A FILE AND PRINTING THE OUTPUT
150 REM INTO A FILE.DATA CAN BE IN A SEQUENTIAL
160 REM FILE OR CAN BE TYPED IN AT RUN TIME. IF THE
170 REM DATA ARE IN A FILE, THE FORMAT IS THE SAME
180 REM AS THAT OF "GRADES" USED IN AVERAGE1 AND
190 REM AVERAGE2. IF THE DATA ARE TO BE TYPED
200 REM IN AT RUN TIME, THEY MUST BE ENTERED
210 REM ACCORDING TO THE SAME FORMAT THEY WOULD
220 REM HAVE WERE THEY IN A FILE.IF OUTPUT IS
230 REM TO GO TO A FILE, THE FILE SHOULD BE SAVED
240 REM BEFORE THE PROGRAM IS RUN.
250 '
270 LET F1 = F2 = 0
280 PRINT "ARE DATA IN A FILE – ANSWER NO OR GIVE
    FILE NAME";
290 INPUT A$
300 IF A$ = "NO" THEN 330
310 OPEN # 1 : FOR INPUT A$
320 LET F1 = 1
330 PRINT "SHOULD OUTPUT GO TO A FILE – ANSWER NO
    OR GIVE"
340 PRINT "FILE NAME";
350 INPUT A$
360 IF A$ = "NO" THEN 390
370 OPEN # 2: FOR OUTPUT A$
380 LET F2 = 2
390 PRINT # F2:
400 PRINT #  F2: "NAME", "AVERAGE"
410 INPUT # F1: S, G
420 PRINT # F2:
430 FOR I = 1 TO S
440 LET A = 0
450 INPUT # F1 : N$
460 FOR J = 1 TO G
470 INPUT # F1 : X
480 LET A = A + X
490 NEXT J
500 LET A = A/G
510 PRINT # F2 : N$,A
520 NEXT I
530 END
```

The following is a sample run of AVERAGE3 using the option to input the data at run time. This listing shows clearly the correspondence between the simple INPUT statement and the INPUT # statement.

```
AVERAGE 3

ARE DATA IN A FILE – ANSWER NO OR GIVE FILE NAME?NO
SHOULD OUTPUT GO TO A FILE – ANSWER NO OR GIVE
FILE NAME? AVERAGES
? 3,4
? GERALD FRIEND
? 78
? 86
? 61
? 90
? PHILIP CLOUGH
? 66
? 87
? 88
? 91
? ADA SHAW
? 56
? 77
? 81
? 85

READY
```

Note that AVERAGE3 will execute as in the example above supplying the file name, TERMINAL, in the first question.

5.2.4 *Other Input/Output Statements*

The LINPUT # statement is used to read strings which might contain such special characters as quotation marks, leading blanks, ampersands, and commas from sequential files. The format of this statement is:

```
100 LINPUT # <expression> : <list of string variables>
```

Rules governing the use of the LINPUT statement apply to the LINPUT # statement.

As we have seen, the INPUT statement requires a comma or carriage return as delimiter for the data being entered into a BASIC program. Because the PRINT statement, in its turn, does not supply the necessary commas, BASIC will not be able to read its own output. This fact has lead to the implementation of the WRITE statement whose purpose is to produce a list readable by a matching INPUT statement. Thus, commas are automatically inserted between the items output. This feature, however, is meaningless when not using files. The format of the statement is:

10 WRITE # <expression> : <list of variables>

There are also five MAT statements which may be used with sequential files: MAT PRINT #, MAT WRITE #, MAT PRINT USING #, MAT INPUT #, and MAT LINPUT #. These statements are discussed in Chapter 6.

5.2.5 *Margins on Sequential Files*

MARGIN # <expression> : <expression>

MARGIN # N : M sets a margin of M on file # N just as the simple MARGIN statement sets a margin on lines output to the terminal. The margin for sequential files may be changed at any time. MARGIN # 0 : M has the same effect as MARGIN M. The interpretation of the margin setting is the same as the simple MARGIN statement. See Section 4.7.7 for details.

5.2.6 *The IF END Statement*

IF END # <expression> THEN <line number>

This statement is similar to ON ERROR GOTO, but has effect only when end of file conditions occur. It must be executed after the OPEN statement and before any INPUT statement reading the actual file. The IF END statement itself is, in fact, no conditional statement at all. When executed the line number is stored in the file table telling BASIC to start the user written error routine if end of the actual file is detected.

The error handling routine can be disabled by executing IF END . . THEN 0. IF END has the highest priority used together with ON ERROR GOTO.

Example: (next page)

```
10 OPEN # 1: FOR INPUT "XXXX"
20 OPEN # 2: FOR INPUT "YYYY"
30 IF END #1 THEN 1000
40 IF END # 2 THEN 2000
50 INPUT # 1,X: INPUT # 2,Y : GOTO 50
60 STOP
1000 REM HERE IF END # 1
1010 IF END # 1 THEN 0
1020 INPUT # 2,X : GOTO 1020
1030 STOP
2000 REM HERE IF END # 2
2010 IF END # 2 THEN 0
2020 INPUT # 1,X : GOTO 2020
3000 END
RUN

BASIC RUN ERROR          3 IN LINE 2020
END OF FILE

READY
```

5.2.7 *Simulating Sequential Files*

BASIC allows *all* input/output statements to communicate with internal strings rather than sequential files. This means that it is possible to convert the numeric value of any expression to an ASCII string or vice versa, according to the rules of the respective input/output statements. Previously we have seen the connect device identifier having numeric values. You will obtain the effects described above if the connect device identifier is given a string value. The general form is:

1. < input statement> # <string expression> : <list of variables>

2. < output statement> # <string variable> : <list of expressions>

The string denoting the connect device identifier is now a BASIC string which is used directly and *not* the name of a sequential file. The OPEN, CLOSE and MARGIN statements have, of course, no meaning in such constructions. Note that output terminates if the stardard margin (75) is exceeded.

If you want to use the numeric value of the substring in A$ starting in position X, and ending in position Y, just type the statement:

```
10 INPUT # SEG$ (A$, X, Y): VALUE
```

On the other hand, if you want to generate a string of the value of A using a special format described in A$, type the statement:

```
10 PRINT USING # FORMAT$ : A$, A
```

If V is a vector and M is a matrix, the entries of V are printed in rows with five entries per row. M is printed as a matrix with the entries of each row closely packed.

Only array names without parantheses are legal in a MAT PRINT statement. The following statements are illegal:

```
100  MAT PRINT M(2,3)
110  MAT PRINT TRN(A)
```

Vectors as well as matrices may be output in the MAT PRINT USING statement. Comma is the only legal delimiter of the format string and the array names in the list. The elements of the array(s) are printed according to the format string as with the PRINT USING statement. The format is used again starting on a new line if there are more elements than fields. If there are several arrays in the list, a blank line is left between them, and the format string is used from the beginning. The shorthand MAT USING may be used,

Example:

```
10  MAT A = CON(2,2)
20 MAT USING "+### AND -#.##↑↑↑↑",A
30  END
RUN
  +1 AND   1.00 E+00
  +1 AND   1.00 E+00
READY
```

6.6.2 *The MAT INPUT and MAT LINPUT Statements and the NUM Function*

The input is taken from the terminal as with normal INPUT or LINPUT statements, and a question mark is printed when the program is ready to accept the input.

If MAT INPUT goes to a vector, the excess data are ignored when trying to enter more data than the vector can hold. If less data are entered, the elements not affected remain unchanged. The function NUM is available after the execution, and returns the number of data which were input.

If MAT INPUT goes to a matrix, the data is entered by row. A variable number of data may not be input; enough data must be entered to fill entirely the matrix as it has been dimensioned in MAT INPUT or previously. The excess data is ignored as with vectors, and the number of data is available in the function NUM.

If you want to input more numbers than can be typed on one line, it is possible to continue typing on additional lines. If the last item on a line is followed by an ampersand (&) with no preceding comma and then by a carriage return, BASIC will accept the input typed so far, and then expect data continued on the following line. The last string on a line must be enclosed in quotation marks if its last character is an ampersand (&).

The following program will call for the input of 24 numbers.

```
100  DIM M(2,12)
110  MAT INPUT M
```

Changing line 110 the program will call for the input of maximum 50 numbers.

```
110  MAT INPUT M(50)
```

String vectors and matrices may also be used in the MAT INPUT statement, and NUM is updated.

The LINPUT statement is described in Section 4.8.1; the MAT LINPUT statement allows more than one line of information (possibly containing commas, leading blanks, etc.) to be input in response to a single statement.

A variable amount of input is *not* allowed, and a question mark is printed for each element.

Common to MAT INPUT and MAT LINPUT is:

- Row 0 and column 0 are ignored.
- Several arrays may appear in the list.
- Arrays may be explicitly redimensioned.
- If not, the current dimension(s) will determine the maximum number of elements to be input.
- Insertion of messages in the list is not allowed as with INPUT and LINPUT.

Examples:

```
100  DIM V(5), A(3), M(3,4)
110 MAT INPUT V, A(2), M(2,3)
120  PRINT "NUM=";NUM
130 MAT PRINT V;A;M;
140  END
RUN
?1,2&
3
?1,2
?1,2,3,4
?4,5,6
NUM= 6
  1    2    3    0    0

  1    2

  1    2    3
  4    4    5
```

```
10 MAT LINPUT A$(4)
20  PRINT "NUM=";NUM
30 MAT PRINT A$
40  END
RUN
?FIRST
?SECOND, (NEXT EMPTY)
?
?FOURTH
NUM= 4
FIRST
SECOND, (NEXT EMPTY)

FOURTH
```

6.6.3 *The MAT WRITE Statement*

As described in Section 5.2.4 the WRITE statement produces an output readable by a matching INPUT statement. The MAT WRITE statement outputs the elements of a vector separated by commas on a single line. The rows of a matrix are output on separate lines, thus readable by a matching MAT INPUT statement. It is very important, however, that the number of characters output on one line does not exceed the margin. This will be dependent on the number of columns and the range of each element. In fact, this restriction is due to the size of the input buffer rather than the current margin.

6.6.4 *MAT Statements and Files*

Any MAT statement performing input or output operations on the terminal may be used with sequential files as well. The formats of the statements are:

```
10  MAT INPUT # <N>:<list of arrays>
20  MAT LINPUT # <N>:<list of string arrays>
30  MAT PRINT # <N>:<list of arrays>
40  MAT USING # <N>:<list of arrays>
50  MAT WRITE # <N>:<list of arrays>
```

where <N> is the connect device identifier; i.e., the number of the file being read or written, or the string which simulates a sequential file.

For a complete discussion of files see Chapter 5.

7.9 *STAND ALONE EXECUTION*

Previously we have seen that any program unit written in BASIC can be compiled to machine instructions in BRF format. Such a program unit is not dependent on being loaded and executed with the total BASIC system in memory. Other subsystems exist which are able to perform the loading and linking procedure:

- SINTRAN III Real Time Loader
- NORD–10/ND–100 Relocating loader

These are described in the respective manuals.

A *BASIC Library and Run-time System* is available for stand alone execution purposes. This system should be loaded after the BASIC program units, hence, only the run-time routines required (called for) are loaded into memory.

7.10 *Mixing BASIC With Other Languages*

BASIC/FORTRAN/NPL/MAC program units, i.e., programs, subroutines or functions may be mixed in an arbitrary combination. — Within the BASIC system at most one BASIC program unit can be executed in incremental mode, else all the units must be compiled to BRF format and linked together by the BASIC built-in loader or by another loader subsystem. The main program may be created in either of the languages mentioned above.

7.10.1 *BASIC Strings as Parameters*

When using a BASIC string as parameter, generally the address of the two word string-descriptor is transferred to callee. The descripton contains the string address (1. word) and string length in bytes (2. word). The string is packed two by two characters in a word.

If, however, a BASIC string appears as parameter to a FORTRAN subprogram, it must be preceded by a dummy plus sign (+). As an effect of this the string address instead of the descriptor address is transferred to callee. This restriction is necessary as the string concept of BASIC is lacking in FORTRAN.

Assignment to string parameters in non-BASIC subprograms will often fail. Such variables should be declared in the COMMON storage area.

Example:

```
10 CALL SUBR1(A$)  'BASIC/BASIC
20 CALL SUBR1(+A$) 'BASIC/FORTRAN
```

On the other hand, a FORTRAN Hollerith string may be associated with a BASIC formal parameter by applying a certain function upon it like:

STRING(<hollerith string>,<number of characters>)

Example:

```
10 CALL SUBR2("ABC")  'BASIC/BASIC
C  FORTRAN/BASIC
   CALL SUBR2(STRING(3HABC, 3))
```

A.2 *RUN–TIME SYSTEM ERROR MESSAGES*

Run-time error messages are printed as self-explanatory text. Example:

BASIC RUN ERROR IN LINE 10: PARITY ERROR ON INPUT

When executing "stand alone" the messages are given as an error code which is an octal number, and the line number is replaced with the octal address of the statement. Numbers in the range 0–377 are equivalent to the error codes returned from the FILE SYSTEM monitor calls. All numbers from 400 and upwards are BASIC run-time error codes which are explained below. FILE SYSTEM errors are always printed with explanatory text in addition to the error code. The ON ERROR GO TO statement will omit printing of run-time error messages, but the error code is still available in the function ERR. In incremental mode errors are always printed with explanatory text. In BRF–compiler mode the user may prevent text strings being loaded (from BASLIBR) if the symbol 7ERRP is set to zero by the DEFINE command prior to loading. If text strings are not loaded, a saving of approx. 1K of memory is achieved.

Error Code Octal	Decimal	Non-fatal (x)	Interpretation
401	257		System error in I/O system
402	258		Format parameter not string
403	259		Illegal delimiter
404	260		Empty string
405	261		Illegal item type
406	262		Out of data
407	263		Not used
410	264		Format error
411	265		System error in I/O system
412	266	x	Integer overflow on input Argument set to largest integer
413	267		Not used
414	268		Input buffer overflow
415	269		Not used
416	270	x	Parity error on input. The character is skipped.
417	271		Bad character on input
420	272		String input error
421	273		Not used
422	274	x	Real overflow on input Argument set to largest real (1E99)

Error Code Octal	Decimal	Non-fatal (x)	Interpretation
423	275	x	Real underflow on input Argument set to zero
424	276	x	Real underflow on output Argument set to zero
425	277	x	Real overflow on output Argument set to largest real (1E99)
426	278		Not used
440	288		Empty or too long string
441	289		Illegal connect device number
442	290		Connect device number used before
443	291		Open-file table filled
444	292		No such connect device number
445	293		Zero or negative margin
446	294		Not used
460	304	x	Overflow in integer exponentiation Result set to largest integer (32767)
461	305	x	Overflow in real–integer exponentiation Result set to largest real (1E99)
462	306	x	Base less than zero in real exponentiation Result set to zero
463	307	x	Overflow in real exponentiation Result set to largest real (1E99)
464	308	x	Argument negative in SQR Result set to zero
465	309	x	Argument overflow in SIN Result set to zero
466	310	x	Argument overflow in COS Result set to zero
467	311	x	Overflow in EXP Result set to largest real (1E99)
470	312	x	Argument zero or negative in LOG/LOG10 Result set to –1E99
471	313	x	Argument error in CAX Argument set to zero

Error Code Octal	Decimal	Non-fatal (x)	Interpretation
472	314	x	Argument overflow in TAN Result set to zero
473	315	x	Overflow in division Result set to zero
474	316	x	Zero base or negative exponent in double integer exponentation. Result set to largest integer.
475	317	x	Argument error in ASI, ACO. Result set to zero.
476	318		Not used
500	320		Double integer in MAT arithmetic statement.
501	321		Dimension unmatch right of = in MAT + or –
502	322		Not used
503	323		System error in MAT * or INV
504	324		Not used
505	325		Dimension unmatch right of = in MAT*
506	326		Dimension error in MAT TRN or IDN
507	327		MAT A = TRN(A) not allowed
510	328		Both arrays must be square in MAT INV
511	329		Both arrays must be two-dimensional in MAT INV
512	330		Both arrays must be real in MAT INV
513	331		Not used
514	332		Dimension out of range
515	333		Argument error in SEG$
516	334		MAT A = A*A not allowed
517	335		Argument error in MATCH
520	336		Argument error in CNT
521	337		Argument error in INS$
522	338		Argument error in REP$
523	339		Argument error in MAXI or MINI
524	340		Not used

Error Code Octal	Decimal	Non-fatal (x)	Interpretation
550	360		GOSUB stack filled
551	361		GOSUB stack empty
552	362		Number of parameters not matching in "FN functions"
553	363		Parameter unmatch in "FN functions"
554	364		"FN stack" filled
555	365		"FN stack" empty
556	366		Statement removed or missing in GOTO/GOSUB
557	367		Statement removed or missing in "FN functions"
560	368		Garbage collection error
561	369		Garbage collection error, out of memory space
562	370		Garbage collection error
563	371		Garbage collection error
564	372		Argument out of range in ON GOTO/GOSUB
565	373		Too many subprograms
566	374		Chaining requires BASIC Compiler
567	375	x	Over/underflow in real addition
570	376	x	Over/underflow in real subtraction
571	377	x	Over/underflow in real multiplication
572	378	x	Overflow in real to integer conversion

If the third parameter is present the compiler will translate the source program into BRF format which is written on the file/device specified. Normally the third parameter is left out indicating incremental operating modus.

In incremental mode the compiled program will be appended to the statements already present (if any).

CONTINUE

The execution of the current program will continue following a STOP statement or a break state.

DEFAULT-INTEGER

All variables will become type INTEGER if not explicitly declared as another type. All constants not including a decimal point or exponent are compiled into single or double integers.

DEFAULT-REAL

Initial modus.

DEFINE <symbol><octal value>

The symbol will be entered into the external-entry-table, its value will be equal to the octal number specified.

DELETE <line number> or <line number–line number>

Remove one or more lines from the current program. Following the word DELETE the user types the line number of the single line to be deleted or two line numbers separated by a dash (–) indicating the first and last line of the section of code to be removed. If the dash is included and the second argument is omitted, the last line of the program is assumed. Several single lines or line sections can be indicated by separating the line numbers, or line number pairs, with a comma. Note that deletion of lines does not remove belonging variables or referenced entry points.

DEPOSIT <octal address>

The old contents of the octal address specified (octally and symbolically) are displayed and may be changed by typing the new contents on the same line. By typing carriage return the next location will be displayed automatically. Termination character is point (.) followed by carriage return.

EDIT <line number>

This command copies the actual line to old line preparing a modification of the line. The line edit control characters may now be applied.

ENTRIES-DEFINED [<file name>]

All symbols (defined) present in the external-entry-table will be printed on the terminal. In addition the current location and the upper bound are displayed in the following format:

FREE: <current location> – <upper bound>

Default file name is the terminal.

ENTRIES-UNDEFINED [<file name>]

This command is much alike ENTRIES-DEFINED, but only undefined symbols are printed.

Default file name is the terminal.

EXIT

Same as BYE.

FIX

The current contents of the external-entry-table are fixed (will not be removed by CLEAR) and the current location will later act as the lower bound reset-address. The fixed entries do not appear in any entry list-out.

IDENTIFIERS-USED [<file name>]

All identifiers used in the current program will be listed on the terminal. Also some type information is given. Default file name is the terminal.

IGNORE-MATRIX-CHECK

Normally, if a matrix is accessed beyond its range (greatest index permitted) a message will be printed. This command removes this checking. Note that a matrix check introduces much overhead as code is generated to compute and check the index(es) for any array access. Should be used for debugging purposes only. Note that this command does not concern COMMON and virtual arrays.

LIBRARY

In this mode subroutines and functions are compiled into library-subprograms. Such subprograms are loaded only if they are referenced from another routine, else they are skipped.

LIST [<line number–line number>]

Produces a listing at the user terminal of the current program, or one or more lines of that program. The word LIST by itself will cause the listing of the entire user program. LIST followed by one line number will list that line; and LIST followed by two line numbers separated by a dash (–) will list the lines between and including the lines indicated. If the dash is included and the second argument is omitted, the last line of the program is assumed. Several single lines or line sections can be indicated by separating the line numbers, or line number pairs, with a comma.

LISTH [<line number> or <line number-line number>]

Same as LIST, but also prints a header containing the program name and current date.

LOAD <file name>[<file name>...]

The file(s) specified will be loaded until EOF (control byte 23) is encountered. The file(s) must be BRF object file(s).

NEW [<program name>]

The BASIC system is initialized and the user may type a new program from his terminal. The command may be followed by a program name (see LISTH and RUNH).

NEXT-LINE

The next line after the last one listed will be printed on the terminal.

N100–REAL–OVERFLOW–CHECK

Turns on/off this check in the compiled code. Overflow as well as underflow is detected in real arithmetic operations in the NORD–100, and an error message is printed (non-fatal). This option is initially turned off.

OBLIST

Special command for system debugging purposes only.

OLD <file name>

The BASIC system is initiated and the program on the file specified will be read and compiled.

RECOMPILE

The source program is re-compiled from its internal scratch file representation. The statements are compiled in ascending order: thus, this command may be the only way to get rid of MISPLACED STATEMENT error messages.

Also the code which belongs to removed or edited statements will disappear.

RENUMBER [<new initial line number> <increment>]

Changes the statement line numbers and the references to these line numbers. First parameter indicates the new initial line number, and the second (if any) indicates the increment in the line numbers of two successive statements. If no parameters are specified the first statement number will be 100 and the increment will be 10.

RUN

Starts execution of the current program.

RUNH

Same as RUN, but also prints a header containing the program name and current date.

Extended Library	Function	Explanation
x	A=POA(X,Y)	returns polar angle of the cartesian coordinates X and Y
x	R=POR(X,Y)	returns polar radius of the cartesian coordinates X and Y
x	Y=FIX(X)	returns the truncated value of X; SGN(X)*INT(ABS(X))
x	Y=FRA(X)	returns the fractional part of X
x	Y=MAXI(A,B,C...)	returns the greatest value
x	Y=MINI(A,B,C...)	returns the smallest value

String Functions

Extended Library	Function	Explanation
	I%=ASC(A$)	returns the ASCII value of the first character in A$
	I%=LEN(A$)	returns the number of characters (bytes) in A$
	A$=SEG$ (B$,F%,L%)	returns a substring of B$ starting in position F% and ending in position L%
	A$=CHR$(X)	returns a one character string (ASCII) corresponding to the value of X
	A$=OC$(I%%)	returns an eleven character digit string corresponding to the value of I%% (octal)
x	N%=CNT(A$,B$)	returns the number of times the string B$ occurs in A$
x	X$=INS$(A$, B$,I%)	returns a string where the contents of the string B$ is inserted into the string A$ at the character position I%.
x	N%=MATCH(A$, B$,I%)	searches the string A$ for the occurrence of the string B$, starting at the I%'th character, The returned value is 0 if no occurrence found, or the position of the first character that match.

Extended Library	Function	Explanation
x	X$=REP$(A$, B$, I%)	returns a string where the string A$ is replaced with the content of the string B$, starting from the I%'th position of the string A$.
x	X$=SPAC$ (I%)	returns a string of spaces, I% characters long

Miscellaneous Functions

Extended Library	Function	Explanation
	TAB(X)	PRINT statements only! Moves print head to position X in the current print record.
	N%=MAR(I%)	returns the last MARGIN setting of connect device no. I%.
	N%=POS(I%)	returns the current print position of connect device no. I%.
	MAT Y=TRN(X)	returns the transpose of the matrix X
	MAT Y=(V)*X	scalar multiplication of each element in matrix X
	MAT Y=INV(X)	returns the inverse of matrix X
	Y=DET	returns the determinant of the last INV(X) function evaluation.
	Y=NUM	returns the number of data input in an array by the last MAT INPUT statement.
	Y=RND	returns a random number between 0 and 1.
	Y=ERR	returns the last error code if an ON ERROR GOTO statement occurs in the program.

APPENDIX C

MISCELLANEOUS INFORMATION

C.1 *ROUNDOFF ERRORS*

The smallest number BASIC can handle is approximately 1*10↑–4931 and the largest number is 1*10↑+4931, but input and output are restricted to be within the following limits: 1*10↑–100< |x|<1*10↑100.

BASIC stores numbers correctly to approximately nine significant digits and generally prints numbers to six significant digits.

The values of the expressions in the FOR or REPEAT statements need not be integers. However, the user must be cautioned that using a non-integer step size may result in roundoff errors. These errors occur because the computer can only store about nine significant digits for each number it computes. The cumulative effect of these roundoff errors over a loop executed many times may be significant:

```
100  FOR X = 0 TO 200 STEP 0.001
110  LET Y = Y+1
120  REM Y COUNTS THE NUMBER OF TIMES
130  REM THE LOOP IS EXECUTED
140  NEXT X
150  PRINT X,Y
160  END
```

This program gave the following output when it was run:

```
200          199998

READY
```

Note that Y, which counts the number of times the loop is performed, is not 200001, the expected value, but 199998; the loop has been executed three times less than might be expected. Consequently, calculations involving the running variable or depending on the number of times the loop was performed would be in error because of roundoff errors.

Thus, in general, use integer step sizes and integer FROM and TO elements to avoid roundoff errors. If you want to step over a series of non-integer values, appropriate operations may be performed on the running variable within the loop to achieve this result. For instance, in the example above X may be made to range from 1 to 200 in steps of .001 using the following technique:

```
100  FOR I = 0 TO 200000
110  LET X = I/1000
120  LET Y = Y+1
130  NEXT I
140  PRINT X,Y
150  END
```

This program prints a value of 200 for X and 200001 for Y. These values are the expected ones, and no roundoff error has occurred.

C.2 *CHANGING DIMENSIONS*

The DIM statement is used to dimension (reserve initial space for) subscripted variables. Thus, the same DIM may be executed in a loop with variable(s) indicating the dimension(s), or the same array may be referenced in separate DIM statements with different dimensions.

Subscripts may be enclosed in parentheses following some MAT statements as follows (one or two dimensions may be specified for all but the IDN function, where two identical values are required).

Functions	Statements
MAT A = CON (N,M)	MAT INPUT A (N,M)
MAT A = IDN (N,N)	MAT LINPUT A (N,M)
MAT A = ZER (N, M)	MAT READ A (N,M)

The array A takes on the dimensions specified in the statement.

Redimensioning is implicit in the MAT statements which perform matrix arithmetic and matrix functions. That is, in the statement MAT C = A+B, C takes on the dimensions of A and B if unequal.

Note that redimensioning (even reservation of less space) is very time-consuming as it involves release of old space and reservation of new space which is always zeroed.

C.3 *LINE EDIT CONTROL CHARACTERS*

The Line Edit control characters available in BASIC are listed below, and on the following pages they are given a short description. (The characters are the same as in SINTRAN III command input.)

Function	Character
Tab	I^C
Line Terminate	M^C (CR)
Escape Character	
take C literally	$V^C C$
Backspace	
one character	A^C
one word	W^C
one line	Q^C
Copy	
one character	C^C
to tab stop	U^C
to end of line	H^C
up to C	$O^C C$
through C	$Z^C C$
rest of line (terminate)	D^C
rest of line (no printing)	F^C
Skip	
one character	S^C
up to C	$P^C C$
through C	$X^C C$
Reprint	
fast	R^C
aligned	T^C
Re-Edit	Y^C
Mode Change	
insert/replace	E^C
terminate	L^C

Graphic	Octal Value	Decimal Value	ASC Abbreviation	Comments
)	51	41	)	Closing parenthesis
*	52	42	*	Asterisk
+	53	43	+	Plus
,	54	44	,	Comma
–	55	45	–	Hyphen (Minus)
.	56	46	.	Period (Decimal)
/	57	47	/	Slant
0	60	48	0	Zero
1	61	49	1	One
2	62	50	2	Two
3	63	51	3	Three
4	64	52	4	Four
5	65	53	5	Five
6	66	54	6	Six
7	67	55	7	Seven
8	70	56	8	Eight
9	71	57	9	Nine
:	72	58	:	Colon
;	73	59	;	Semi-colon
<	74	60	<	Less than
=	75	61	=	Equals
>	76	62	>	Greater than
?	77	63	?	Question mark
@	100	64	@	Commercial at
A	101	65	A	Uppercase A
B	102	66	B	Uppercase B
C	103	67	C	Uppercase C
D	104	68	D	Uppercase D
E	105	69	E	Uppercase E
F	106	70	F	Uppercase F
G	107	71	G	Uppercase G
H	110	72	H	Uppercase H
I	111	73	I	Uppercase I
J	112	74	J	Uppercase J
K	113	75	K	Uppercase K
L	114	76	L	Uppercase L
M	115	77	M	Uppercase M
N	116	78	N	Uppercase N
O	117	79	O	Uppercase O
P	120	80	P	Uppercase P
Q	121	81	Q	Uppercase Q
R	122	82	R	Uppercase R
S	123	83	S	Uppercase S

Graphic	Octal Value	Decimal Value	ASC Abbreviation	Comments:
T	124	84	T	Uppercase T
U	125	85	U	Uppercase U
V	126	86	V	Uppercase V
W	127	87	W	Uppercase W
X	130	88	X	Uppercase X
Y	131	89	Y	Uppercase Y
Z	132	90	Z	Uppercase Z
[	133	91	[	Opening bracket
\	134	92	\	Reversing slant
]	135	93	]	Closing bracket
^	136	94	^	Circumflex, up-arrow
_ or ←	137	95	_,UND,BKR	Underscore, back arrow
`	140	96	`, GRA	Grave accent
a	141	97	a, LCA	Lowercase a
b	142	98	b, LCB	Lowercase b
c	143	99	c, LCC	Lowercase c
d	144	100	d, LCD	Lowercase d
e	145	101	e, LCE	Lowercase e
f	146	102	f, LCF	Lowercase f
g	147	103	g, LCG	Lowercase g
h	150	104	h, LCH	Lowercase h
i	151	105	i, LCI	Lowercase i
j	152	106	j, LCJ	Lowercase j
k	153	107	k, LCK	Lowercase k
l	154	108	l, LCL	Lowercase l
m	155	109	m, LCM	Lowercase m
n	156	110	n, LCN	Lowercase n
o	157	111	o, LCO	Lowercase o
p	160	112	p, LCP	Lowercase p
q	161	113	q, LCQ	Lowercase q
r	162	114	r, LCR	Lowercase r
s	163	115	s, LCS	Lowercase s
t	164	116	t, LCT	Lowercase t
u	165	117	u, LCU	Lowercase u
v	166	118	v, LCV	Lowercase v
w	167	119	w, LCW	Lowercase w
x	170	120	x, LCX	Lowercase x
y	171	121	y, LCY	Lowercase y
z	172	122	z, LCZ	Lowercase z
{	173	123	{ , LBR	Opening (left) brace
|	174	124	|, VLN	Vertical line
}	175	125	} , RBR	Closing (right) brace
~	176	126	~, TIL	Tilde
	177	127	DEL	Delete, rubout