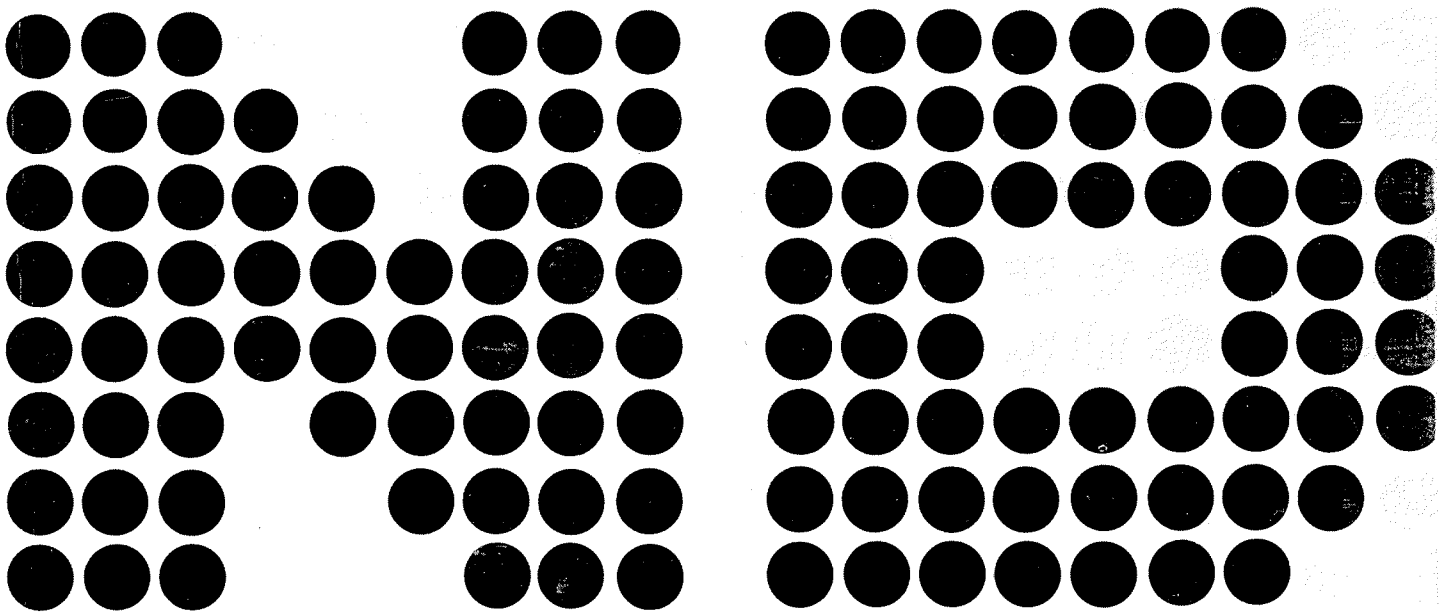
NORD-10 BASIC
Compiler
Reference Manual

NORSK DATA A.S



NORD-10 BASIC
Compiler
Reference Manual

REVISION RECORD

Revision	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.
	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.

A/S NORSK DATA-ELEKTRONIKK
Lørenveien 57, Oslo 5 - Tlf.: 21 73 71

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 INTRODUCTION

1.1 *WHAT IS A COMPUTER?*

A computer is a very simple and at the same time a very complex machine. On the one hand, it merely follows elementary instructions to carry out such simple tasks as adding two numbers or determining if a given number is negative. These simple tasks also include "looking" at the next character in a string of alphabetic characters and other non-numeric activities.

On the other hand, a modern electronic digital computer must be surrounded by a number of storage devices and input-output mechanisms which supply it with tasks to perform, store the results of its computations, and present these results in a convenient form for evaluation or future use. A computer performs its work so fast that these peripheral devices are needed to correlate the many tasks the computer is capable of performing.

1.2 *WHAT IS A PROGRAM?*

As noted above, a computer merely carries out simple instructions, albeit at very high speeds. It works so quickly that human beings cannot be directly involved in making more than a small fraction of the decisions that arise in carrying out a complicated task, so that almost all situations must be contemplated in advance. Also, in most cases, the bulk of the data upon which the calculations are made must be accurately prepared in advance and entered into the computer so that the calculations may proceed at full speed without having to wait for more data. Thus, a set of instructions for performing a task and the relevant data must be prepared in advance and supplied to the computer. The set of instructions for carrying out a task is called a *program*. One can think of a program as being a recipe for coming up with the solution to a problem, given the data.

Any mistakes in a program render it just about useless. As with recipes for baking cakes, program errors are of two types. First, one can have errors of form or grammar. These would include misspellings and punctuation. Second, one can have substantive errors even though the form is correct. In the case of recipes for baking cakes, misspelling and typographical errors are examples of errors of form; some of these may make the recipe unreadable. An example of a substantive error would be a direction to use baking soda instead of baking powder.

Since a computer has much less intelligence or common sense than a human being, programs for it must adhere strictly to rules of form or grammar. These rules are particularly complicated for the language that the physical equipment of the computer is constructed to obey. This language is called *machine language*, and its difficult nature has led computer specialists to invent other more easily used languages that can be converted or translated to machine language.

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 the 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.

2 A BASIC PRIMER

2.1 *AN EXAMPLE*

The following example is a complete BASIC program for solving a system of two simultaneous linear equations in two variables:

$$ax + by = c$$
$$dx + ey = f$$

and then solving two different systems, each differing from this system only in the constants c and f.

You should be able to solve this system, if ae – bd is not equal to 0, to find that:

$$x = \frac{ce - bf}{ae - bd} \qquad \text{and} \qquad y = \frac{af - cd}{ae - bd}$$

If ae – bd = 0, there is either no solution or there are infinitely many, but there is no unique solution, If you are rusty at solving such systems, take our word for it that this is correct. At the moment, we want you to understand the BASIC program for solving the system.

Study this example carefully – in most cases the purpose of each line in the program is self-evident – and then read the commentary and explanation.

```
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
```

We immediately observe several things about this sample program. First, we see that the program uses only capital letters, since the terminal has only capital letters.

A second observation is that each line of the program begins with a number. These numbers are called *line numbers* and serve to identify the lines, each of which is called a *statement*. Thus a program is made up of statements, most of which are instructions to the computer. Line numbers also serve to specify the order in which the statements are to be performed by the computer. This means that you may type your program in any order. Before the program is run, the computer sorts out and edits the program, putting the statements into the order specified by their line numbers. This editing process facilitates the correcting and changing of programs, as we shall explain later.

A third observation is that each statement starts, after its line number, with an English word. This word denotes the type of the statement. There are several types of statements in BASIC; some of them are discussed in this chapter.

A fourth observation, not at all obvious from the program, is that spaces have no significance in BASIC, except in messages enclosed in quotation marks which are to be printed out, as in line number 65 on the previous page. Thus, spaces may be used, or not used, to "pretty up" a program and make it more readable. Statement 10 could have been typed as 10READ A, B, D, E and statement 15 as 15LET G= A*E–B*D.

With this preface, let us go through the example step by step. The first statement, 10, is a READ statement. It must be accompanied by one or more DATA statements. When the computer encounters a READ statement while executing your program, it will cause the variables listed after the READ to be given values according to the next available numbers in the DATA statements. In the example we read A in statement 10 and assign the value 1 to it from statement 70 and similarly with B and 2, and with D and 4. At this point, we have exhausted the available data in statement 70, but there is more in statement 80, and we pick up from it the number 2 to be assigned to E.

We next go to statement 15, which is a LET statement, and first encounter an expression to be evaluated. (The asterisk "*" is obviously used to denote multiplication.) In this statement we direct the computer to compute the value of AE – BD, and to call the results G. In general, a LET statement directs the computer to set a variable equal to the expression on the right side of the equal sign. We know that if G is equal to zero, the system has no unique solution. Therefore, we next ask, in line 20, if G is equal to zero. If the computer discovers a "yes" answer to the question, it is directed to go to line 65 where it prints "NO UNIQUE SOLUTION". From this point it would go to the next statement. But lines 70, 80 and 85 give it no instructions, since DATA statements are not "executed", and it then goes to line 90 which tells it to "END" the program.

If the answer to the question "Is G equal to zero?" is "no", as it is in this example, the computer goes on to the next statement, in this case 30. (Thus an IF–THEN tells the computer where to go if the "IF" condition is met, but to go on to the next statement if it is not met.) The computer is now directed to read the next two entries from the DATA statements, –7 and 5, (both are in statement 80) and to assign them to C and F respectively. The computer is now ready to solve the system

$$x + 2y = -7$$
$$4x + 2y = 5$$

In statements 37 and 42 we direct the computer to compute the value of X and Y according to expressions provided. Note that we must use parentheses to indicate that CE – BF is divided by G; without parentheses, only BF would be divided by G and the computer would let X = CE – BF/G.

The computer is told to print the two values computed, that of X and that of Y, in line 55. Having done this, it moves on to line 60 where it is directed back to line 30. If there are additional numbers in the DATA statements, as there are here in 85, the computer is told in line 30 to take the next one and assign it to C, and the one after that to F. Thus, the computer is now ready to solve the system:

$$x + 2y = 1$$
$$4x + 2y = 3$$

As before, it finds the solution in 37 and 42 and prints them out in line 55, and then is directed in 60 to go back to 30.

In line 30 the computer reads two more values, 4 and –7, which it finds in line 85. It then proceeds to solve the system:

$$x + 2y = 4$$
$$4x + 2y = -7$$

and to print out the solutions. It is directed back again to 30, but there are no more pairs of numbers available for C and F in the DATA statement. The computer then informs you that it is out of data, printing a message on your terminal.

Run time errors are errors detected during execution of a program whereas errors detected during compilation of a program are called compile time errors. A complete error list is given in Appendix A.

For a moment, let us look at the importance of the various statements. For example, what would have happened if we had omitted line 55? The answer is simple; the computer would have solved the three systems and then told us when it was out of data. However, since it was not asked to tell us (PRINT) its answers, it would not do it, and the solutions would be the computer's secret. What would have happened if we had left out line 20? In this problem just solved nothing would have happened. But, if G were equal to zero, we would have set the computer the impossible task of dividing by zero, and it would tell us so, printing a warning on your terminal. If we had left out statement 60, the computer would have solved the first system, printed out the values of X and Y, and then gone to line 65 where it would be directed to print "NO UNIQUE SOLUTION". It would do this and then stop.

One very natural question arises from the seemingly arbitrary numbering of the statements: Why this selection of line numbers? The answer is that the particular choice of line numbers is arbitrary, as long as the statements are numbered in the order we want the machine to follow in executing the program. We could have numbered the statements 1, 2, 3, 4, ...13, although we do not recommend this numbering. We would normally number the statements 10, 20, 30, ... 130. We put the numbers a certain distance apart so that we can later insert additional statements if we find that we forgot them when we originally wrote the program. Thus, if we find that we have left out two statements between those numbered 40 and 50, we can give them any two numbers between 40 and 50 – say 44 and 46; and in the editing and sorting process, the computer will put them in their proper place.

Another question arises from the seemingly arbitrary placing of the data elements in the DATA statements: Why were they placed as they were in the sample program? Here again the choice is arbitrary and we need only to put the numbers in the order that we want them read (the first for A, the second for B, the third for D, the fourth for E, the fifth for C, the sixth for F, the seventh for next C, etc.). In place of the three statements numbered 70, 80 and 85, we could have put:

```
75 DATA 1, 2, 4, 2, –7, 5, 1, 3, 4, –7
```

or we could have written, perhaps more naturally:

```
70 DATA 1, 2, 4, 2
75 DATA –7, 5
80 DATA 1, 3
85 DATA 4, –7
```

to indicate that the coefficients appear in the first data statement and the various pairs of right-hand constants appear in the subsequent statements.

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.

2.4 *ARRAYS OR MATRICES*

In addition to the ordinary variables used by BASIC there are variables which can be used to designate the elements of an array. These are used where we might ordinarily use a subscript, for example, the coefficients of a polynomial (a_0, a_1, a_2 . . . . . .) or the elements of a matrix ($B_{i,j}$). The variables which we use in BASIC are called the name of the array, followed by the subscript(s) in parenthesis. Thus, we might write A(0), A(1), A(2), etc., for the coefficients of the polynomial and B(1,1), B(1,2), etc., for the elements of the matrix. In this manual you will also find the words dimension or index for subscript and indexed variable for subscripted variable.

We can enter the array A(0), A(1), A(2), ... A(10) into a program very simply by the lines:

```
10 FOR I = 0 TO 10
20 READ A (I)
30 NEXT I
40 DATA 2, 3, –5, 2.2, 4, –9, 123, 4, –4, 17
```

We need no special instruction to the computer if no subscript greater than 10 occurs. However, if we want larger subscripts, we must use a DIM statement to indicate to the BASIC system that it has to save extra space for the array. When in doubt, indicate a larger dimension than you expect to use. For example, if we want a list of 15 numbers entered, we might write:

```
10 DIM A (25)
20 READ N
30 FOR I = 1 TO N
40 READ A (I)
50 NEXT I
60 DATA 15
70 DATA 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47
```

Statements 20 and 60 could have been eliminated by writing 30 as FOR I = 1 TO 15, but the form as typed would allow for the lenghtening of the array by changing only statement 60, as long as it did not exceed 25.

We would enter a 3 x 5 array into a program by writing:

```
10 FOR I = 1 TO 3
20 FOR J = 1 TO 5
30 READ B (I, J)
40 NEXT J
50 NEXT I
60 DATA 2, 3, –5, –9, 2
70 DATA 1, –7, 3, 4, –2
80 DATA 3, –3, 5, 7, 8
```

Here again, we may enter an array with no dimension statement, and it will handle all the entries from B(0,0) to B(10,10). If you try to enter an array with a subscript greater than 10, without a DIM statement, you will get an error message telling you that you have a subscript error. This is easily rectified by entering the line:

```
5 DIM (20, 30)
```

if, for instance, we need a 20 by 30 matrix.

The form of the subscript is quite flexible, and you might have the array element B(I, K) or Q(A(3,7), B–C).

Shown below is a list and run of a problem which uses both a singly and a doubly subscripted array. The program computes the total sales of each of five salesmen, all of whom were selling the same three products. The array P gives the price/item of the three products and the array A tells how many items of each product each man sold. You can see from the program that product number 1 sells for $1.25 per item, number 2 for $4.30 per item, and number 3 for $2.50 per item; and also that salesman number 1 sold 40 items of the first product, 10 of the second, and 35 of the third, and so on. The program reads in the sales array in lines 40 – 80, using data in lines 910 – 930. The same program could be used again, modifying only line 900 if the price changes, and only lines 910 – 930 to enter the sales in another month.

This sample program did not need a dimension statement, since the computer automatically saves enough space to allow all subscripts to run from 0 to 10. A DIM statement is normally used to save more space. However, in a long program, requiring many small arrays, DIM may be used to save less space for arrays, in order to leave more for the program.

Since the DIM statement is used to save space for arrays, the DIM statement must be executed before the space is being used. Normally the DIM statements will be placed near the beginning of the program.

```
 10 FOR I = 1 TO 3
 20 READ P(I)
 30 NEXT I
 40 FOR I = 1 TO 3
 50 FOR J = 1 TO 5
 60 READ A (I, J)
 70 NEXT J
 80 NEXT I
 90 FOR J = 1 TO 5
100 LET S = 0
110 FOR I = 1 TO 3
120 LET S = S + P(I) * A (I, J)
130 NEXT I
140 PRINT "TOTAL SALES FOR SALESMAN"; J; "$"; S
```

```
150 NEXT J
900 DATA 1.25, 4.30, 2.50
910 DATA 40, 20, 37, 29, 42
920 DATA 10, 16, 3, 21, 8
930 DATA 35, 47, 29, 16, 33
999 END

RUN
TOTAL SALES FOR SALESMAN 1 $180.5
TOTAL SALES FOR SALESMAN 2 $211.3
TOTAL SALES FOR SALESMAN 3 $131.65
TOTAL SALES FOR SALESMAN 4 $166.55
TOTAL SALES FOR SALESMAN 5 $169.4

READY
```

2.5 *USE OF THE SYSTEM*

Commands to the computer, unlike statements or instructions in a program have no line number. These commands are typed at the start of a new line on the terminal and are followed by a carriage return. They give the computer information on manipulating programs you are creating or have previously written. You may type a command any time you could type a numbered line in a program.

Execution of a program is started by the command RUN, and you can obtain a listing in its current form of your program by typing the command LIST. If you wish to save it for later use, type SAVE. To recover a program previously saved, type OLD and then the program name. The result will be just as if you had typed in a new program and saved it.

Now that we know something about writing a program in BASIC, how do we set about using a terminal to type in our program and then have the computer solve our problem?

First, ascertain that the BASIC system is present. If no, the system is loaded as explained in Section 3.1. When the computer types READY you should begin to type your program. Make sure that each line begins with a line number which contains no non-digit characters. Be sure to press the carriage return key at the completion of each line. Spaces may be inserted at any point in the line, including before the line numbers.

If, in the process of typing a statement, you make a typing error and notice it immediately, you can correct it by pressing A^C. (Press the key marked CTRL and type A). This will delete the preceding character and you can then type in the correct character. Pressing this key a number of times, for example n, will erase from this line the n last characters. To delete all of the present line, press Q^C. Programs or data may be annotated by typing the remark and then deleting the line (as far as the system is concerned) with Q^C. BASIC prints "←" to show that a line has been deleted, and "↑" to show that a character has been deleted.

When a line is finished, you press the carriage return key. Then the statement is analyzed by the computer and if any syntax error is found, an error message is printed.

After typing your complete program, you type RUN, press the carriage return key and hope. If the program is one which the computer can run, it will then run it and print out any results for which you have asked in your PRINT statements. This does not mean that your program is correct, but that it has no errors of the type known as "grammatical errors". If it had errors of this type, the computer would have printed an error message as soon as the error was detected during the typing of the program. Errors detected after RUN are structural (loop nesting, matching GOSUB and RETURN) or arithmetical errors. A complete error list is given in Appendix A together with the interpretation of each.

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.

Before going to the terminal, we write a program, let us assume that it is the following:

```
 10 READ D
 20 LET X0 = 0
 30 FOR X = 0 TO 3 STEP D
 40 IF SIN (X)< = M THEN 100
 50 LET X0 = X
 60 LET M = SIN (X0)
 70 PRINT X0, X, D
 80 NEXT X0
 90 GO TO 20
100 DATA . 1, .01, .001
110 END
```

We shall list the entire sequence on the terminal and make explanatory comments.

```
NEW MAXSIN

 10 REAF D
***ERROR IN LINE 10 NOT RECOGNIZED
 10 READ D
 20 LET X0 = 0
 30 FOR X = 0 TO 3 STEP D
 40 IF SINE↑   ( X )< = M THEN 100
 50 LET X0 = X
 60 LET M = SIN  ( X )
 70 PRINT X0,X,D
 80 NEXT Z↑X0
 90 GOTO 20
100 DATA .1,. 01, .001
110 END
RUN
"LINE 80" IMPROPERLY NESTED FOR LOOPS
```

A message indicates that READ was mistyped in line 10, so we retype it, this time correctly.

Notice the use of A^C (types up arrow) to erase a character in line 40, which should have started IF SIN (X) etc., and in line 80.

Upon checking we see that the variable in the FOR and NEXT are different, so we correct statement 80. In looking over the program, we also notice that the IF - THEN statement in 40 directed the computer to a DATA statement and not to line 80 where it should go.

```
80 NEXT X
40 IF SIN (X)< = M THEN 80
RUN
0.1          0.1          0.1
0.2          0.2          0.1
0.3          0.3          0.1
USER BREAK AT 110334
@CONT
READY
```

M has never been assigned an initial value and is assumed to be zero. We decide to give it a value less than the maximum value of the sine, say -1.

```
20 LET M= -1
RUN
0            0            0.1
0.1          0.1          0.1
0.2          0.2          0.1
USER BREAK AT 110334
@CONT
READY
```

This is incorrect. We are having every value of X0,X and the interval size printed, so we direct the machine tc cease operations by depressing the ESC-key even while it is running. Notice that the ESC does not print, but the word USER BREAK is printed.

We fix this by moving the PRINT statement outside the loop. We also realize that we want M printed and not X.

```
DELETE 70
85 PRINT X0, M, D
RUN
1.6          9.99574E-01          0.1
1.6          9.99574E-01          0.1
USER BREAK AT 110334
@CONT
READY
```

Of course, line 90 sent us back to line 20 to repeat the operation and not back to line 10 to pick up a new value of D. We also decide to put in headings for our columns by a PRINT statement.

```
90 GOTO 10
5 PRINT "X VALUE", "SIN", "RESOLUTION
***ERROR IN LINE 5 ILL. STRING TERMINATION
```

There is an error on our PRINT statement, no right quotation mark for the third item.

Retype line 5 by typing H^{U} and supply the missing quotation mark.

```
5 PRINT "X VALUE", "SIN", "RESOLUTION"
RUN
X VALUE        SIN           RESOLUTION
1.6            9.99574E-01   0.1
1.57           1             0.01
1.571          1             0.001
BASIC RUN ERROR 406 IN LINE 10
READY
```

Exactly the desired results. Of the 31 numbers (0, .1, .2, .3, ... 2.8, 2.9, 3) it is 1.6 which has the largest sine, namely .999574. Similarly for finer subdivisions.

Having changed so many parts of the program, we ask for a list of the corrected program. Listing the corrected program, from time to time, is an important part of debugging. Using LISTH will list the program name as a header, including the date and time:

```
LISTH

MAXSIN   15 JULY  1976    09.31.59
5 PRINT "X VALUE", "SIN", "RESOLUTION"
10 READ D
20 LET M = -1
30 FOR X = 0 TO 3 STEP D
40 IF SIN(X) <= M THEN 80
50 LET X0 = X
60 LET M = SIN(X)
80 NEXT X
85 PRINT X0, M, D
90 GO TO 10
100 DATA .1, .01, .001
110 END

SAVE "MAXSIN"
```

The program is saved for later use by writing it on the file MAXSIN. A file name enclosed in double quotes means that a new file with the given file name should be created.

In solving this problem, there is a common device which we did not use, namely the insertion of a PRINT statement when we wonder if the machine is computing what we think we asked it to compute. For example, if we wondered about M, we could have inserted 65 PRINT M and we would have seen the values.

With more difficult problems we can use the STOP statement, 58 STOP.

However, it is more convenient to use the BREAK debugging option which allows to set a break point at the beginning of any statement by typing BREAK followed by the actual statement number. A breakpoint halts the execution and returns control to the terminal. All legal statements executing in immediate mode (statement without statement number) may now help us to examine and change the variable values if necessary.

It is also possible to change the program. If wanted, set a new breakpoint before going on with the command CONTINUE.

2.6.1 *Use of Flags*

The technique of ending a program by having it run out of data is very simple and efficient. However, it does not yield an attractive printout and prevents taking any action *after* the program discovers that it has run out of data. The MAXSIN program above terminates with a run error message telling it is out of data in line 10.

Now that we have the IF–THEN statement, we can agree that a 0 signifies the end of the data. In reading the data, when we reach the 0, we will know that all computations have been done and we can now terminate the execution. The number 0 is used as a piece of data having special meaning and is called a *flag*. Actually, we can agree to any number as the data-ending flag, but we chose 0 because a step size of 0 never occurs.

In the MAXSIN example we should make the following corrections:

```
 15 IF D = 0 THEN 110
100 DATA .1, .01, .001, 0
```

2.7 *SUMMARY OF ELEMENTARY BASIC STATEMENTS*

In this section we shall give a short and concise description of each of the types of BASIC statements discussed earlier in this chapter and add some to our list.

2.7.1 *LET*

LET <variable> = <expression>

This statement is not a statement of algebraic equality, but rather a command to the computer to perform certain computations and to assign the answer to a certain variable.

Examples:

```
100 LET X = X + 1
259 LET W7 = (W - X4↑3)*(X - A/(A - B))-17
```

2.7.2 *READ and DATA*

READ <list of variables>
DATA <list of numbers>

We use a READ statement to assign to the listed variables values obtained from a DATA statement. Neither statement is used without one of the other type. A READ statement causes the variables listed in it to be given, in order, the next available numbers in the collection of DATA statements. Before the program is run, the computer takes all DATA statements in the order in which they appear and create a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data, with a READ statement still asking for more, it is assumed that the program is done and we get an out-of-data error code.

Since we have to read in data before we can work with it, READ statements normally occur near the beginning of a program. The location of DATA statements is arbitrary as long as they occur in the correct order. A common practice is to collect all DATA statements and place them just before the END statement.

Examples:

```
150 READ X, Y, Z, X1, Y2, Q9
330 DATA 4, 2, 1.7
340 DATA 6.734E-3, -174.321, 3.14159265
```

```
234 READ B (K)
263 DATA 2, 3, 5, 7, 9, 11, 10, 8, 6, 4

 10 READ R (I, J)
440 DATA -3, 5, -9, 2.37, 2.9876, -437.234E-5
450 DATA 2.765, 5.5576, 2.3789E2
```

Remember that only numbers are put in a DATA statement and that 15/7 and $\sqrt{3}$ are expressions not numbers.

2.7.3 *PRINT*

PRINT <list of expressions>

The PRINT statement has a number of different uses and is discussed in more detail in Section 4.7. The common uses are:

1. To print out the result of some computations
2. To print out verbatim a message included in the program
3. A combination of the two
4. To skip a line

We have seen examples of only the first three in our sample programs. Each type is slightly different in form, but all begin with PRINT after the line number.

Examples of type 1:

```
100 PRINT X, SQR (X)
135 PRINT X, Y, Z, B * B–4 * A * C, EXP(A–B)
```

The first will print X and then a few spaces to the right of that number its square root. The second will print five different numbers:

X, Y, Z, B^2 –4AC, and e^{A-B}

The computer will compute the two expressions and print them for you. It can print up to five numbers per line in this format.

Examples of type 2:

```
100 PRINT "NO UNIQUE SOLUTION"
430 PRINT "X VALUE", "SINE", "RESOLUTION"
```

Both have been encountered in the sample programs. The first prints that simple statement, the second prints the three labels with spaces between them. The labels in 430 automatically line up with the three numbers called for in a PRINT statement (as long as the labels do not exceed 14 characters) as seen in MAXSIN.

Examples of type 3:

```
150 PRINT "THE VALUE OF X IS"; X
 30 PRINT "THE SQUARE ROOT OF"; X; "IS"; SQR(X)
```

If the first has computed the value of X to be 3, it will print out: THE VALUE OF X IS 3 . If the Second has computed the value of X to be 625, it will print out: THE SQUARE ROOT OF 625 IS 25. The semi-colon delimiter will be discussed later.

Examples of type 4:

```
250 PRINT
```

The computer will advance the paper one line when it encounters this command.

2.7.4 *GOTO*

GOTO <line number>

There are times in a program when you do not want all statements executed in the program. An example of this occurs in the MAXSIN problem where the computer has computed X0, M, and D and printed them out in line 85. We did not want the program to go to the END statement yet, but to go through the same process for a different value of D. Therefore, we directed the computer to go back to line 10 with a GOTO statement. (It is possible to go to a non-executable statement, control then passes to the sequential executable statement.)

Example:

```
150 GO TO 75
```

2.7.5 *IF-THEN- or IF-GOTO*

IF <expression> <relation> <expression> GOTO <line number>

There are times when we are interested in jumping the normal sequence of statements, if a certain relationship holds. For this we use an IF – THEN statement, sometimes called a conditional GO TO statement. Such a statement occurred at line 40 of MAXSIN.

Examples:

```
40 IF SIN (X) < = M THEN 80 or
40 IF SIN (X) < = M GO TO 80

20 IF G = 0 THEN 65 or
20 IF G = 0 GO TO 65
```

The first asks if the sine of X is less than or equal to M, and directs the computer to skip to line 80 if it is. The second asks if G is equal to 0, and directs the computer to skip to line 65 if it is. In each case, if the answer to the question is no, the computer will go to the next line of the program.

2.7.6 *FOR and NEXT*

```
FOR <variable> = <expression> TO <expression> [STEP
    <expression>]
NEXT <variable>
```

We have already encountered the FOR and NEXT statements in our loops, and have seen that they go together, one at the entrance to the loop and one at the exit, directing the computer back to the entrance again.

Any simple (not subscripted) variable may be used as the FOR variable. Most commonly, the expressions will be integers, and the STEP omitted. In the latter case, a step size of one is assumed. The accompanying NEXT statement is simple in form, but the variable must be precisely the same as that following FOR in the FOR statement.

Examples:

```
 30 FOR X = 0 TO 3 STEP D
 80 NEXT X
120 FOR X4 = (17 + COS(Z))/3 TO 3*SQR(10) STEP 1/4
235 NEXT X4
456 FOR J = -3 TO 12 STEP 2
```

Notice that the step size may be an expression: (1/4), or a positive number(2). In the example with lines 120 and 235, the successive values of X4 will be .25 apart, in increasing order. In the last example, on successive trips through the loop, J will take on values -3, -1, 1, 3, 5, 7, 9 and 11.

Note that variables implied in final or step size expression plus the control variable itself can be changed in the body of the loop. Thus the loop control may be affected during execution.

If you write 50 FOR X = 2 TO –2, the body of the loop will not be performed and the computer will proceed to the statement immediately following the corresponding NEXT statement.

Negative STEP elements permit running through a sequence of values "backwards." For instance:

```
320 FOR X = 10 TO 0 STEP –3  'Must be a negative constant'
```

will cause X to run through the values 10, 7, 4 and 1.

2.7.7 *DIM*

DIM <variable name (dimension(s)),. . .>

As we have seen BASIC needs no information concerning the array size if the subscript(s) run from0 to 10. (The default size is then automatically set up by BASIC.) In those circumstances when a one-dimensional array has more than 11 entries, or when a two-dimensional array has more than 11 rows or 11 columns, we specify the needed size of the array in a DIM statement.

```
20 DIM A(20), B(2, 15), C(4), D(10)
```

The array named A has entries numbered from 0 through 20, B has rows numbered 0 through 2 and columns numbered 0 through 15, C has five entries numbered 0 through 4, and D has entries numbered from 0 through 10. Note that C is dimensioned smaller than default to save space in the computer. D could have been left out since the dimension is equal to the default. Here is some rules concerning the use of DIM and subscripted variables:

1. DIM statements may appear anywhere in a program. All array elements are set to zero after a DIM statement is executed. String elements are empty.

2. The same name denoting a variable and an array will lead to conflicts.

3. The dimension(s) of an array may be any legal expression. This means that arrays can be re-dimensioned during run time if the DIM statement is executed in a loop, for instance. It is even possible to re-dimension a one-dimensional array to be two-dimensional or vice versa.

4. A two-dimensional array can be accessed with one dimension.

5. If the evaluated expression describing the subscript(s) is not an integer value, it will be truncated to the nearest integer. Note that the evaluation is truncation, not rounding.

6. The total index is limited to 65535 (a 16 bit data word).

2.7.8 *STOP*

A STOP statement may be entered anywhere in a program. With STOP, execution is halted and control is passed to the terminal.

Example:

```
25 STOP
```

2.7.9 *END*

Every program must have an END statement and it must be the statement with the highest line number in the program. When the computer reaches the END statement the execution of the program stops.

Example:

```
999 END
```

2.7.10 *ON-GOTO*

ON <expression> GOTO <list of line numbers>

Using an IF – THEN statement provides only a two-way branch in a program. A decision between only two alternatives can be made. More branches can be achieved by using multiple IF – THEN statements. However, a single statement, ON – GOTO, allows a manyway branch.

For example, the following lines in a longer program:

```
 90 READ X
100 IF X = 1 THEN 500
110 IF X = 2 THEN 600
120 IF X = 3 THEN 700
130 DATA 3
```

could be replaced by these three lines:

```
 90 READ X
100 ON X GO TO 500, 600, 700
130 DATA 3
```

Example:

```
100 ON X GO TO 500, 600, 700
```

If X is equal to 1, the computer takes its next instruction from line 500, if X is 2, control passes to 600, and so on. If the value of X is not an integer, its integer part is used. If the value of X is less than one or greater than the number of line numbers listed, an error messages is given. There may be any number of line numbers listed in the instruction as long as the entire instruction fits on a single line.

2.7.11 *REM and Remarks*

REM <text>

REM provides a means for inserting explanatory remarks in a program. The system completely ignores the remainder of that line, allowing the programmer to follow the REM with directions for using the program, with identifications of the parts of a long program, or with anything else that he wants. Although what follows REM is ignored, its line number may be used in a IF – THEN, GO TO, or ON – GO TO statement.

Explanatory remarks may be located following a statement on a line, by using the character '. Anything on the line following ' will be treated as an explanatory remark. For example, the statement

```
250 LET Y = 1 ' INITIALIZE Y TO ONE
```

includes the remark INITIALIZE Y TO ONE without affecting the running of the program.

In line 130 the line number is followed by an apostrophe and the rest of the line is left blank. Such blank lines are used to increase the readability of the program listing.

Example:

```
100 REM INSERT DATA IN LINES 900 - 998. THE FIRST
110 REM NUMBER IS N, THE NUMBER OF POINTS. THEN
120 REM THE DATA POINTS THEMSELVES ARE ENTERED
130 '
     :
200 IF N >< 0 GOTO 500
     :
     :
500 REM N WAS UNEQUAL ZERO
     :
     :
```

2.7.12 *RESET*

Sometimes it is necessary to use the data in a program more than once. The RESET statement permits reading the data as many additional times as it is used. Whenever RESET is encountered in a program, the system resets the data block pointer to the first number. A subsequent statement will then start reading the data all over again. A word of warning – if the desired data is preceded by code numbers or parameters, superfluous READ statements should be used to pass over these numbers. For example, the following program portion reads the data, restores the data block to its original state, and reads the data again. Note the use of line 570 to "pass over" the value of N which is already known.

```
100 READ N
110 FOR I = 1 TO N
120 READ X
     :
     :
200 NEXT I
     :
     :
560 RESET
570 READ X
580 FOR I = 1  TO N
590 READ X
```

2.7.13 *INPUT*

INPUT <list of variables>

There are times when it is desirable to have data entered during the run of a program. This is particularly true when one person writes the program and enters it into memory, and other persons are to supply the data. This may be done by an INPUT statement which acts as a READ statement but does not draw numbers from a DATA statement. If, for example, you want the user to supply values for X and Y into a program, you will type

```
40 INPUT X, Y
```

before the first statement which is to use either of these numbers. When it encounters this statement, *the system will print a question mark.* The user types two numbers, separated by a comma, presses the return key and the system goes on with the rest of the program.

A single carriage return means that the line is empty. If there are more numbers or strings in the line than were requested, the excess is ignored. If there are not the same number of items as there are variables in the INPUT list, a new question mark is printed indicating that the program needs more data.

Frequently an INPUT statement is combined with a PRINT statement to make sure that the user knows which values to put in. You might type

```
20 PRINT "WHAT ARE YOUR VALUES OF X, Y, AND Z";
30 INPUT X, Y, Z
40 END
RUN
```

and the system will print

```
WHAT ARE YOUR VALUES OF X, Y, AND Z?
```

Without the semicolon at the end of line 20, the question mark would have been printed on the next line.

Data entered via an INPUT statement are not saved with the program. Furthermore, it may take a long time to enter a large amount of data using INPUT. Therefore, INPUT should be used only when small amounts of data are to be entered, or when it is necessary to enter data during the running of the program such as with a game-playing program.

Input from the terminal also allows the user to insert messages to be printed between the variables to be input. Such strings must always be constants enclosed in quotes.

The statement:

```
10 INPUT "X=", X, "Y=", Y, "Z=", Z
```

will execute like this:

```
X = ?1
Y = ?2
Z = ?3
```

or this:

```
X = ?1, 2
Z = ?3
```

or this:

```
X = ?1, 2, 3
```

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.

Examples:

```
REN 10, 2
RENU 1000
RENUM
```

3.1.3 *Naming of Programs*

Program names are used as a header with listings and runs if the commands RUNH or LISTH are used (H = header).

The program name should start with a letter and have no more than 32 characters. Quotes, spaces and other non-printable characters should not be used.

If you use OLD, the file name is used as program name.

You may set a program name by typing the command NEW followed by the new program name.

```
NEW SQUARES
```

will set SQUARES as the current program name.

NEW without any name will just initialize the system.

3.2 *SAVING AND RETRIEVING BASIC PROGRAMS*

When you are working on a program and want to continue later, you should save the program by using the SAVE command with the appropriate mass storage file name. A hard copy is produced using Teletype or line printer, a tape may be punched using paper punch, etc.

A saved program is entered later using OLD (or COMPILE) followed by the appropriate file name.

3.2.1 *The SAVE Command*

SAVE <file name>

The SAVE command will save a BASIC program. The appropriate names for the SAVE command are as follows:

TERMINAL designates the user terminal
F–P designates the fast punch
L–P designates the line printer

With other file names the system expects that you want the program saved on a mass storage file.

SAVE SQUARE

will save the current program on the mass memory file named SQUARE. If you have no file with such a name, the file must be created. To do this you should enclose the file name in quotes.

SAVE "SQUARE"

will create the file SQUARE and save the current program onto this file.

Program names may be used as file names, with the exception of the names of the standard peripherals. Further information on file naming is found in the documentation of the NORD-10 File System.

3.3 *EXECUTING YOUR PROGRAM*

When you think your program or part of it, is finished, you can try to run it using the command RUN or RUNH.

Before execution starts, the system will reset variable values and check the program. If no errors are found, program execution is started. Execution will continue until either STOP or an END statement is found or until fatal error condition occurs. Then execution is terminated, and control is passed to the BASIC command processor.

If a STOP statement is encountered, program execution is halted and you may then examine and change your program. Execution is continued by giving the command CONTINUE.

3.3.1 *The RUN Command*

This command is used to initiate execution. You may type RUNH which means the computer should start by printing the program name as identification. Execution starts with the first line.

Before program execution is started, the system will reset variables and check the program for mismatching FOR – NEXT, errors with multiline DEF FN functions, etc.

String identifiers are regarded as empty.

3.3.2 *Terminating Execution*

Program execution is continued until an END statement is reached, an error is found or you break execution by typing the break character (ESC).

Possibly your program will produce erroneous results or it may be executing some endless loops. You can then force execution to be terminated by depressing the ESC-key.

When you press the ESC push button the SINTRAN III command processor is entered and you may restart BASIC by typing CONTINUE. Restarting after an ESC will usually be successful, but may fail under some circumstances.

3.3.3 *Immediate Mode Execution*

Most statements permitted in the NORD BASIC System may be used without line numbers preceding them. In this form they are treated as immediate commands and executed directly rather than being appended to a program.

Thus typing

```
PRINT 0.5, SIN (0.5)
```

will result in the output

```
0.5             4.79426E-01
```

Immediate mode is especially powerful while debugging programs by printing the values of certain identifies and/or assigning to them new values. Also, some small loops may be executed directly in immediate mode with the aid of a multiple statement line (Sections 4.14.1 and 4.14.2).

Example 1:

An infinite loop printing the square root of a specified argument:

```
REPEAT@: INPUT X: PRINT SQR (X)
?16
4
?

etc.
```

Note: This program can be terminated only by depressing the ESC-key.

Example 2:

Printing a table of 2^X while X ranges from 1 to 10.

```
 REPEAT 10: PRINT "2**";@,2**@
2**1            2
2**2            4
2**3            8
2**4            16
2**5            32
2**6            64
2**7            128
2**8            256
2**9            512
2**10           1024

READY
```

Certain statements have no meaning when presented in immediate mode, thus the following ones are not permitted:

DEF	FNEND	
DATA	READ	
FOR	NEXT	
PROGRAM	SUBROUTINE	FUNCTION
ON ERROR	RETURN	END

Upon these statement types the system will react with the message:

```
ILL. IN IMMEDIATE MODE
```

3.3.4 *Setting Break Points*

The common debugging method of BASIC programs is to insert STOP statement at certain places (traps). If any of these STOPs are executed the variable values may tell the programmer why the program has come into this state.

Nevertheless, insertions and removals of STOP statements may in some cases represent a cumbersome way of debugging.

NORD-10 BASIC offers an option of setting break points at a specified statement. When a break point is reached the execution will halt and the control is transferred to the BASIC command processor.

This break is performed before the specified statement is executed. In this state a new break may be specified and the execution continues by using the CONTINUE command. The use of break points combined with immediate mode provides a powerful and simple debugging aid. A break point is reset when it is executed.

A break point is specified by

```
BREAK <line number>
```

Example of a program execution with breaks:

```
10 LET A = 1
20 LET A = 2
30 END

BREAK 10
RUN
BREAK IN LINE 10
PRINT A
0
```

```
READY
BREAK 20
CONTINUE
BREAK IN LINE 20
PRINT A
1

READY
CONTINUE

READY
PRINT A
2

READY
```

3.4 *LEAVING THE BASIC SYSTEM*

When you have finished programming and saved what programs you would like to use later, you should type BYE. This command will enter the operating system. The command EXIT is equivalent to BYE.

4 MORE ABOUT BASIC

4.1 *ELEMENTS OF BASIC*

4.1.1 *Constants*

Five types of constants are used in BASIC: Integer, Double Integer, Real, Octal and String. The type of a constant is determined by its form. (If the DEFAULT-INTEGER command is used, see also Section B.2.) The computer word structure for each type is given in Appendix C.5.

Integer

An integer constant consists of up to five decimal digits ended with % in the range of $-2^{15} \leqslant n \leqslant 2^{15} - 1$. An integer constant occupies one word of main memory.

Examples:

63%	–3241%	896%
247%	27963%	–4343%

Double Integer

A double integer constant consists of up to 10 digits ended with % in the range of $-2^{31} = -2147483648 \leqslant n \leqslant 2147483647 = 2^{31} - 1$. A double integer constant occupies two consecutive storage locations if it is outside the range of an integer constant, or forced to double integer by leading zeros:

Example:

–444444%	999000000%	0000000%

Real

Real constants are expressed with or without a decimal point or with a fraction and an exponent representing a power of ten. The form of real constants are:

.nE	.nE±s	n	n.E±s
n.n	n.nE±s	.n	n

n is the base, s is the exponent to the base 10. The plus sign may be omitted for a positive s.

A real constant occupies three (optionally two) consecutive main storage locations.

Examples:

501436	42	104
3.1415768	–314.	.013469
.31416E1	3.14E06	–31.415E-1

Octal

An octal constant is denoted by one to eleven octal digits post-fixed by the letter B. If more than eleven digits are specified, only the last eleven are significant. Octal constants are single or double integers; double if the number does not fit into one word, or forced to by leading zeros.

Examples:

123456B	-7B	177777777B

Note: Octal constants may turn negative if the sign bit is set. 177777B is equal to ÷1B.

String

A string constant consists of any sequence of ASCII characters enclosed in quotation marks. The quotation mark may not be part of the constant since it is used to define the beginning and the end of the string. The size of a string constant is limited by the line length.

Examples:

"OLA" ". . ." "YOU ARE MY SWEETHEART!"

A string constant occupies (n + 1)/2 + 3 memory locations. n is the number of characters in the string.

4.1.2 *Variables*

Variable names are alphanumeric identifiers that represent specific storage locations.

BASIC recognizes simple and subscripted variable names. Default variable type is real unless the DEFAULT-INTEGER command is used (Section B.2). The type of the variable may be defined in a type declaration statement (Section 4.1.3). Otherwise, the type is determined by the postfixed letter(s) of the variable name.

A variable name is a single letter, or a letter followed by a digit, or a letter followed by up to six digits and/or letters. If the name exceeds seven characters, the seven left-most characters will comprise the variable name but the last character(s) will still determine the type of the variable. When using a seven character variable name, the type declaration character is necessary only in the first occurrence of the variable.

Integer

A variable name followed by one % character.

Examples:

I%, I1%, I123%, NUMBER%

Double Integer

A variable name followed by two % characters.

Examples:

DB%%, DB1%%, TALLY%%

Real

A variable name.

Examples:

A, A1, A12345, ABCDEFG

String

A variable name followed by one $ character.

Examples:

A$, A123$, FIELD$, NAME$

Subscripted Integer

A variable name followed by one % character, followed by one or two subscripts within parenthesis.

Examples:

POWER%(5), I%(X%, Y%), I1%(X, 5%)

Subscripted Double Integer

A variable name followed by two % characters, followed by one or two subscripts within parenthesis.

Examples:

DB%%(5), TALLY%%(X,Y)

Subscripted Real

A variable name followed by one or two subscripts within parenthesis.

Example:

A(5), ATOM (X, Y)

Subscripted String

A variable name followed by one $ character, followed by one or two subscripts within parenthesis.

Examples:

A$(5), NAMES%(X, Y)

4.1.3 *Type Declaration Statements*

Statements of this kind are called declarative or non-executable and must be the first statements entered in a BASIC program. Thus, the compiler is provided with information abut the structure of variable or function identifiers. Such declarations overrule the type implied by the identifier name.

Integer

INTEGER <list of variables>

10 INTEGER I, I1, I123, NUMBER

declares the four identifiers to be of integer type, i.e., one word element.

Double Integer

DOUBLE <list of variables>

20 DOUBLE DB, DB1, TALLY

declares the three identifiers to be of double integer type, i.e., two words per element.

Real

REAL <list of variables>
30 REAL A, B, C

declares the three identifiers to be of real type.

4.2 *ARITHMETIC EXPRESSIONS*

String expressions will be discussed later.

BASIC will admit general arithmetic expressions in most connections where numbers are allowed. Exceptions are: line numbers must be positive integers. Numbers are used in data statements and with input.

An arithmetic expression is a constant, variable (simple or subscripted), an evaluated function, or any combination of these separated by arithmetic operators, commas or parentheses to form a meaningful mathematical expression.

4.2.1 *Arithmetic Symbols or Operators*

In the examples in this chapter, arithmetic expressions are used and examples of the way they are evaluated by the computer are given. Five symbols representing arithmetic operations can be used in expressions. These symbols are listed in the table below: the first four are used in the programs in this chapter.

Symbol	Expression	Meaning
+	A + B	Addition: add B to A
–	A – B	Subtraction: subtract B from A
*	A * B	Multiplication: multiply A by B
/	A / B	Division: divide A by B
** or ↑	A ↑ B	Exponentiation: raise A to power B (on many terminals the symbol for exponentiation is Λ.)
–	–A	Unary minus: a minus which starts an expression or which follows immediately after = or (

4.2.2 *Elements*

The elements of arithmetic expressions are formed as follows:

A primary is an arithmetic expression in parenthesis, a constant (positive or zero), variable, array element or function reference:

(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 placement of parentheses in an expression can alter the sequence in which the operations are performed. Two of the preceding examples have been rewritten to illustrate this.

1. 10 ↑ (2 + 1)

   The computer evaluates this expression as 10 ↑ 3 = 1000. The expression inside the parentheses is evaluated first and then the exponentation is performed.

2. ((32 / 4) ↑ 2 + 3) * (3 – 1)

   This expression will be evaluated as follows: (8 ↑ 2 + 3) * (3 – 1) = 67*2 = 134. The expression inside the "innermost" parentheses is evaluated first. Within parentheses the described sequence of performing the operations applies.

Since two BASIC arithmetic operators may not be adjacent, parentheses are needed in some expressions containing negative numbers. For example, "X raised to the –2 power" should be written X ↑ (–2), and "–3 subtracted from 2" should be written 2 – (–3).

In summary, to insure the proper interpretation of expressions you should remember that the computer performs exponentiation first, multiplication and division second and addition and subtraction last unless otherwise indicated by placement of parentheses. When in doubt about how an expression will be evaluated, use parentheses.

4.3 *MIXED MODE ARITHMETIC EXPRESSIONS*

Arithmetic expressions can contain mixed types of constants and variables. Mixed mode arithmetic is often accomplished through the special library conversion subroutines which are a part of BASIC run-time system.

The order of dominance of the operand types within an expression is real-double integer-integer.

In mixed mode arithmetic, the mode used to evaluate any portion of an expression is determined by the dominant type so far encountered within the expression and the normal hierarchy of arithmetic operations: integer mode will be used when an integer type is first encountered and will be converted to real mode when a real type is encountered.

The following table indicates how the mode is determined from the possible combinations of variables.

+ – * /	Integer	Double Integer	Real
Integer	Integer	Double Integer	Real
Double Integer	Double Integer	Double Integer	Real
Real	Real	Real	Real

Examples:

1. Given A, B type real: I, J type integer. The mode of evaluating the expression (A * B – I + J) will be real because the dominant operand is type real. It is evaluated:

   $A * B \rightarrow R_1$ real

   Convert I to real

   $R_1 - I \rightarrow R_2$ real

   Convert J to real

   $R_2 + J \rightarrow R_3$ real

The placement of parentheses in an expression can alter the sequence in which the operations are performed. Two of the preceding examples have been rewritten to illustrate this.

1. 10 ↑ (2 + 1)

   The computer evaluates this expression as 10 ↑ 3 = 1000. The expression inside the parentheses is evaluated first and then the exponentation is performed.

2. ((32 / 4) ↑ 2 + 3) * (3 – 1)

   This expression will be evaluated as follows: (8 ↑ 2 + 3) * (3 – 1) = 67*2 = 134. The expression inside the "innermost" parentheses is evaluated first. Within parentheses the described sequence of performing the operations applies.

Since two BASIC arithmetic operators may not be adjacent, parentheses are needed in some expressions containing negative numbers. For example, "X raised to the –2 power" should be written X ↑ (–2), and "–3 subtracted from 2" should be written 2 – (–3).

In summary, to insure the proper interpretation of expressions you should remember that the computer performs exponentiation first, multiplication and division second and addition and subtraction last unless otherwise indicated by placement of parentheses. When in doubt about how an expression will be evaluated, use parentheses.

4.3 *MIXED MODE ARITHMETIC EXPRESSIONS*

Arithmetic expressions can contain mixed types of constants and variables. Mixed mode arithmetic is often accomplished through the special library conversion subroutines which are a part of BASIC run-time system.

The order of dominance of the operand types within an expression is real-double integer-integer.

In mixed mode arithmetic, the mode used to evaluate any portion of an expression is determined by the dominant type so far encountered within the expression and the normal hierarchy of arithmetic operations: integer mode will be used when an integer type is first encountered and will be converted to real mode when a real type is encountered.

The following table indicates how the mode is determined from the possible combinations of variables.

+ – * /	Integer	Double Integer	Real
Integer	Integer	Double Integer	Real
Double Integer	Double Integer	Double Integer	Real
Real	Real	Real	Real

Examples:

1. Given A, B type real: I, J type integer. The mode of evaluating the expression (A * B – I + J) will be real because the dominant operand is type real. It is evaluated:

   $A * B \rightarrow R_1$ real

   Convert I to real

   $R_1 - I \rightarrow R_2$ real

   Convert J to real

   $R_2 + J \rightarrow R_3$ real

2. The use of parentheses can change the evaluation. A, B, I, J are defined as above. (A * B – (I – J)) is evaluated:

   $A * B \rightarrow R_1$ real

   $I - J \rightarrow R_2$ integer

   Convert R_2 to real

   $R_1 - R_2 \rightarrow R_3$ real

3. The order of the elements in an expression can change the evaluation A, B, I, J are defined as above. The expression (J – I + A + B) is evaluated:

   $J - I \rightarrow R_1$ integer

   Convert R_1 to real

   $R_1 + A \rightarrow R_2$ real

   $R_2 + B \rightarrow R_3$ real

Rules:

1. The order of dominance of the standard operand types within an expression from highest to lowest is:

   REAL
   DOUBLE INTEGER
   INTEGER

2. The mode of an evaluated arithmetic expression is referred to by the name of the dominant operand type.

3. In expressions of the form A**B the following rules apply:

   - B may be negative when the form is A**(–B).

   - For the standard types the mode/type relationships are:

TYPE B

TYPE A	Integer	Double Integer	Real	Mode of A ** B
Integer	Integer	Illegal	Real	
Double Integer	Double Integer	Illegal	Illegal	
Real	Real	Illegal	Real	

4.3.1 *More About LET*

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

In the LET statement, values can be assigned to variables, as with the READ and INPUT statements (eg., 100 LET X = 2). However, the LET statement is also a command to the computer to perform certain computations and to assign the answer to a certain variable (eg., 110 LET X = X + 1).

More generally, several variables may be assigned the same value by a single LET statement. Two examples follow:

```
100 LET X = Y3 = 1E2
110 LET A(X) = X = X + 1
```

Note that in line 110 the index is computed first, i.e., the old value of X is used for the subscript of A. That is, after execution of line 110, A(100) and X are equal to 101 and A(101) remains unchanged. Note also that numeric constants may be represented in scientific notation, as well as in integer or fractional notation, anywhere in the program.

The fact that arithmetic assignment statements appear very frequently in programs has led to the convenience of omitting the word LET from the LET statement. This means that we can write an assignment like this:

```
100 X = X + 1
110 X(1) = X(2) = Y + 3
```

4.3.2 *Mixed Mode and LET Statements*

The general form of the arithmetic assignment statement is

v = e

v is an identifier, e is the evaluated arithmetic expression.

Although the type of an evaluated expression is determined by the type of the dominant operand, this does not restrict the types that the identifier v may assume.

Rules for Assignment for e to v:

v type:	e type:	Assignment:
Integer	Integer	Assign
Integer	Double Integer	Convert double integer to integer and assign
Integer	Real	Fix to integer and assign
Double Integer	Integer	Convert integer to double integer and assign
Double Integer	Double Integer	Assign
Double Integer	Real	Fix to double integer and assign
Real	Integer	Float and assign
Real	Double Integer	Float and assign
Real	Real	Assign

Examples: (Given A type real, I, J type integer)

1. A = I + J is evaluated as:

   I + J → R_1 integer

   Convert R_1 to real

   Store R_1 in A

2. I = J + A is evaluated as:

   Convert J to real

   J + A → R_1 real

   Convert R_1 to integer

   Store R_1 in I

4.4 *ARRAYS*

As we have seen, a subscripted variable may have one or two subscripts. If the subscript has more than two dimensions, an error message is given. Any two dimensional array may also be referred to as if it were a one-dimensional. Subscripts may be constants, variables or expressions of any numeric type, however, non-integer values will delay the execution as all subscripts are evaluated in the specified data type and then converted to integer. A subscripted variable references a single element in an array, the subscripts describe the relative location within the array.

An array is a block of successive memory locations for storage of variables. In certain contexts, the entire array, or sometimes element zero, may be referred to by the array name without subscripts. Each element of an array is referenced separately by the array name plus the subscript notation.

The type of array is determined by the array name or the type declaration. The number of dimensions in an array subscript indicates the dimension of the array, the magnitude of each dimension indicates the maximum value that the subscript may take. Program execution errors may result if subscripts are larger than the dimensions initially declared for the array.

The amount of memory allocated to an array depends on the array type and dimensions.

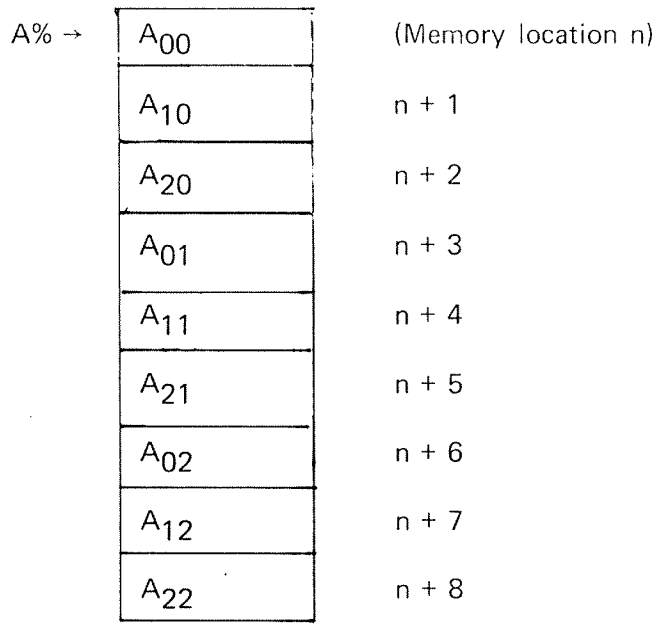
BASIC does not necessarily assign sequential storage to two or more arrays.

4.4.1 *Array Structure*

Elements of arrays are stored by columns in ascending order of storage location. The ordering of elements in an array follows the rule that the first subscript (row) varies most rapidly and the last subscript (column) varies least rapidly. The integer array declared as A%(2, 2), will normally be looked upon as a table consisting of rows (→) and columns (↓), like this:

A_{00}	A_{01}	A_{02}
A_{10}	A_{11}	A_{12}
A_{20}	A_{21}	A_{22}

The layout in memory will be as follows.



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$(4, 5) denotes a string in the 4th row and 5th column of a table of strings B$.

A DIM statement such as

```
100 DIM A$ (25)
```

is required if any subscript will exceed 10. Individual entries of string arrays can be assigned in LET statements as in the following example.

```
220 LET T$ = A$ (J + 1)
230 LET A$ (J + 1) = A$ (J)
```

The individual entries of an array have no limit regarding the string length.

4.6.5 *An Operator for Combining Strings*

One operation has been defined as working specifically on strings. This is concatenation, denoted by the ampersand (&). Concatention puts one string directly after another, without any intervening characters.

Example:

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

Running this program causes "TIMESHARING" to appear on the terminal. It is possible to use string constants in quotation marks in place of string variables with the & operator, if desired.

```
50 PRINT A$ & " " & B$
```

will print A$ and B$ with a blank between them.

4.6.6 *String Expressions*

In the examples above we have seen examples of string expressions which may consist of a constant, variable (simple or subscripted), an evaluated function, or any combination of these separated by the & operator, commas or parentheses.

4.6.7 *Functions Regarding Strings*

Like the mathematical functions, BASIC provides various functions for use with strings. These functions allow the program to access part of a string, determine the number of characters in a string, generate a character string corresponding to a given number or vice versa, search for a substring within a larger string and perform other useful operations.

Converting numbers to strings or vice versa is, in fact, not performed by functions at all, but handled by using the different input/output statements in BASIC. This is described in Section 5.2.7, Simulating Sequential Files.

The four most elementary functions are described below. Other existing string functions are described in Appendix B.3. User defined functions will be described later.

The ASC Function

It is awkward to memorize the correspondence between numbers and graphics defined by the ASCII code. Rather than being forced to remember that A corresponds to 65, the programmer can make use of the ASC function and write ASC ("A").

The function will take a string as a argument and deliver a number as a result. Only the first character of the string is used.

Example:

```
10 PRINT ASC ("A")
```

The LEN Function

The LEN function takes a string as an argument and returns the number of characters as a result.

Example:

```
10 LET DIFF % = LEN (X$) – LEN (Y$)
```

The CHR$ Function

CHR$ (Z) delivers a one-character string which corresponds to the numeric value of the expression Z. According to ASCII code as outlined in Section C.4, the maximum value of Z is normally 127. However, as far as printing graphics is concerned, characters are equivalent modulo 128; that is, the remainder when the number is divided by 128 is used. For example, 511 = 127 modulo 128. So CHR$(511) = CHR$(127). A single line statement which will print a quotation mark follows:

```
100 PRINT CHR$ (42B)
```

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.)

!! — 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.)

III A number printed in *scientific* notation has the following form.

Z E + Y or Z E –Y

where Z is a number greater than 1 and less than 10 printed in fractional notation (II) and Y is the appropriate power of 10.

Numbers are printed in one of these notations according to their magnitude and type. All numbers are rounded off to six significant digits.

1. An integer whose absolute value is less than 10↑6(1000 000) is printed in integer notation (I).

2. An integer whose absolute value is greater than or equal to 10↑6 is printed in scientific notation (III).

3. A number whose absolute value is greater than or equal to .1 and less than 999999, and which is *not an integer* is printed in fractional notation (II).

4. A number whose absolute value is less than .1 which can be expressed using 6 digits after trailing (right-most) zeros are dropped is printed in fractional notation (II).

5. A number whose absolute value is less than 1 which does not satisfy the condition in (4) is printed in scientific notation (III).

6. A number whose absolute value is greater than 999999 and which is *not an integer* is printed in scientific notation (III).

By printing powers of two, the following program illustrates how numbers falling into each of these six categories are printed.

```
100 FOR I = 1 TO 30 STEP 3
110 PRINT 2↑(–I), I, 2↑I
120 NEXT I
130 END
```

This program yields the following printout.

```
0.5          1          2
0.0625       4          16
7.8125E-03   7          128
9.76562E-04  10         1024
1.2207E-04   13         8192
1.52588E-05  16         65536
1.90735E-06  19         524288
2.38419E-07  22         4.1943E + 06
2.98023E-08  25         3.35544E + 07
3.72529E-09  28         2.68435E + 08

READY
```

4.7.6 *The TAB Function*

In addition to the previously described standard means of controlling printing formats, it is possible to set up non-standard columns and to print material in special forms. The TAB functon is one way of producing such specialized output.

If the first column in which information can be printed on the terminal is labeled column 0, then the comma can be thought of as performing a tabulation to the next tab stop; these stops are set at columns 15, 30, 45 and 60. There is a way to tab to any desired column using the TAB function. The TAB function can appear only in a PRINT list. It does not cause anything to be printed but it simply positions the terminal print head to begin printing in the column denoted by the argument of the TAB function. For example:

```
100 PRINT X; TAB (12); Y; TAB (27); Z
```

will cause the X value to be printed starting in column 0, the Y value in column 12 and the Z value in column 27.

The TAB function may contain any expression as its argument. The value of the expression is computed and the integer part is taken. This number is treated modulo the current margin setting to obtain a column number. The terminal print head then spaces to this position; in the event that it has already passed this position the TAB is ignored.

4.7.7 *The MARGIN Statement*

```
MARGIN <expression>
```

The MARGIN statement sets the maximum number of characters which may be printed on a line. The margin is initially set to 75. If the output string is so long that there is not enough room even on a complete single line, as much as will fit will be printed on that line; the rest of the output string will be continued on the next line, and the process will be repeated as many times as necessary to print the entire string. Even if a line is partially filled when a MARGIN statement is executed, the statement will change the margin for the rest of the line.

The program MARGIN is illustrative:

```
10 A$ = "MARGIN"
20 FOR I = 1 TO LEN (A$) STEP 2
30 MARGIN I
40 PRINT A$
50 NEXT I
100 END
```

The output is:

```
M
A
R
G
I
N

MAR
GIN

MARGI
N

READY
```

4.7.8 *The PRINT USING Statement*

[PRINT] USING <string>, <list>

In addition to the standard formats defined above, it is possible to define your own formats and use them. This feature allows you to print numbers in columns so that decimal points line up and to produce tables easily.

Instead of employing

```
100 PRINT A, B, C$
```

you can modify the PRINT statement to

```
100 PRINT USING X$, A, B, C$
```

Here, X$ contains a "picture" of the line to be printed. Spaces where the values of the variables are to be inserted are marked by special conventions. Literal labels may also be part of the picture string. If desired, a string constant may be used in place of X$, and constant information may be printed in place of variables.

A sample use of the PRINT USING statement follows:

```
100 LET A = 20
110 LET B = 15
120 LET C$ = "CDE"
130 LET X$ = "A IS – # , B IS – # ,  AND THE STRING C IS < ##"
140 PRINT USING X$, A, B, C$
150 END
```

When this program is run,

A IS 20, B IS 15, AND THE STRING C IS CDE

appears on the terminal.

There are 8 special characters for defining PRINT USING fields or areas where variables are to be printed. These 8 characters are: #, –, +, ↑ ., $, < and >. The number sign, "#" reserves a place for one character in a field, but it cannot be used at the beginning of a field.

The effect of these characters is summarized in the following chart:

Sign Valid In	Effect
– Numeric fields only	Start field; print floating minus for negative numbers *.
+ Numeric fields only	Start field; print floating plus or minus as appropriate *.
. Numeric fields	Mark decimal place
$ Numeric fields only	Start field; print floating dollar sign: must be followed by + or –.
↑ Numeric fields only	Specify exponent field: must be in group of 5.
< String fields only	Start field; print string left-justified.
> String field only	Start field; print string right-justified.
# Any field	Place holder.

* These characters may be immediately preceded by "$".

A numeric field, an area in which a number is to be printed, begins with either "–" or "+". If "+" is used, a plus or minus sign will be printed just before the left-most digit of the number, depending on whether it is positive or negative. If "–" is used, there will be a sign only before a negative number. A "–" alone can be used to specify a one character numeric field; a non-negative number less than 10 may be printed using such a format. Additional places in a numeric field can be specified by repeating "#" as many times as desired.

Numbers are rounded and truncated before they are printed. They are printed right-justified in the field, so that the integer digits line up on successive lines. A sample program NUMBERS1 is:

```
100 PRINT USING "LINE 100 - ###", 200.34
110 PRINT USING "LINE 110 - ###", 20.03
120 PRINT USING "LINE 120 - ###", 2.00
130 END
```

Output from a run of NUMBERS1 follows:

```
LINE 100      200
LINE 110       20
LINE 120        2
```

If a number has too many digits to be printed in the field given, asterisks are supplied instead. So with the format "– #", the number "200" appears as "**" on the terminal when the statement is executed: the field " – #" could be used to print numbers in the range $-10 < X < 100$. If a field has more places than there are significant digits allowed in BASIC, question marks are supplied for the digits which might be misleading. In an eleven space field, a number input as "1111111111" is printed as "111111111?",

If numbers are not to be printed as integers, a period is used to mark the location of the decimal point in the numeric field. (A . is interpreted as a character to be printed literally if it is not in a numeric field.) If the number "20.356" is printed with the format "– ##.##" two decimal places are given, the number is rounded and truncated accordingly and the result is printed as "20.36". Again, the number is right-justified in the field so that the decimal points line up on successive lines. For numbers in the range $-1 < X < +1$, a leading zero is provided. As an example, consider the program NUMBERS2 as follows:

```
100 PRINT USING "LINE 100 - ## . ##", 20.356
110 PRINT USING "LINE 110 - ## .## ", 2.0356
120 PRINT USING "LINE 120 - ##. ##",  .20356
130 PRINT USING "LINE 130 - ##. ## ",  -0.5
```

A run of NUMBERS2 produces the following output:

```
LINE 100      20.36
LINE 110       2.04
LINE 120       0.20
LINE 130       *.**
```

To print a number with an exponent, put a group of five up-arrows (the symbol for exponentiation) into the format string; the count of 5 is mandatory. If 2.356 is printed with the format "– # . # ↑↑↑↑↑", "2.4 E + 00" appears. With the ↑ format, one space is reserved for a possible sign and the number begins with the next space. The exponent is adjusted to compensate for any shifting which occurs. With a field "– ### . ##↑↑↑↑↑", the number 2.356 appears as "235.60 E – 02", and the number 20.356 is printed "203.56 E–01".

The following program NUMBERS3 exemplifies these conventions:

```
100 PRINT USING "LINE 100 –## . ##↑↑↑↑↑, 203.56
110 PRINT USING "LINE 110 –## . ##↑↑↑↑↑, 20.356
120 PRINT USING "LINE 120 –## . ##↑↑↑↑↑, 2.0356
130 PRINT USING "LINE 130 –## . ##↑↑↑↑↑, .20356
140 END
```

Running this program gives the following output:

```
LINE 100 20.36 E + 01
LINE 110 20.36 E + 00
LINE 120 20.36 E – 01
LINE 130 20.36 E – 02
```

An exception to the rule that a numeric field must begin with a "+" or "–" is the option of preceding these two characters by a "$". The use of a dollar sign forces the printing of a dollar sign just before the first digit or sign of the number.

It is possible to have literal information, including commas, in a format field. In particular, it is possible to include blanks to group digits conveniently. With the format string "–##, ### . ##", "99999" will be printed as "99,999.00". Since a field must begin with – + $ < or >, it is possible to interrupt it with literal information. This literal information must not include any of the special characters, except that a period in a non-numeric field is printed literally.

The field for printing a string must begin with either < or >. These characters are valid only in string fileds, just as +, –, ↑. and $ are valid only in numeric fields. A < causes the string to be printed left-justified in the field specified. If necessary, the field is filled with blanks or the string truncated from the right. As with numeric fields, "#" serves to hold a place for printing. Left-justification of strings is shown in the following example; program STRINGS1:

```
100 PRINT USING "LINE 100 < ##", "AB"
110 PRINT USING "LINE 110 < ##", "ABC"
120 PRINT USING "LINE 120 < ##", "ABCD"
130 END
```

Running this program gives:

```
LINE 100 AB
LINE 110 ABC
LINE 120 ABC
```

A > sign causes the string to be printed right-justified in the field specified. If necessary, the string is preceded by enough blanks to fill the field or is truncated from the left. Altering the last program to STRINGS2:

```
100 PRINT USING "LINE 100 > ##", "AB"
110 PRINT USING "LINE 110 > ##", "ABC"
120 PRINT USING "LINE 120 > ##", "ABCD"
130 END
```

we get

```
LINE 100      AB
LINE 110      ABC
LINE 120      BCD
```

Again, literal information can be included within the field; with "<#X##", the string "ABCD" is printed as "ABXCD".

Note that it is not possible to specify any of the special characters # – + ↑ $ < or > as material to be printed literally. If these special characters are to appear in the output, they can be specified as constants to be printed in separate fields. To print a "+", the following statement suffices.

```
900 PRINT USING "<", "+"
```

The items to be printed according to the defined format must be separated by commas and a comma must separate the USING string from the variables. The order of numeric and string variables to be printed must match the order of the types in the format string. For example:

```
900 PRINT USING "–# . ## < ###", "ABCD", 23.4
```

causes an error message and termination of the execution because the field types in the format string do not match the types of information to be printed. A string cannot be printed with a numeric field nor can a number be printed with a string field.

If there are fewer variables in the list of a PRINT USING statement than there are fields specified in the format string, the extra fields are not used. On the other hand, if there are more variables than fields, the format string is used again, starting on a new line. If the information to be printed will not fit on a single line, the part of the format not used on the first line is counted on the second line, and so on until all the items in the list are printed.

Ending a PRINT USING statement with a semicolon causes suppression of the carriage return and line feed characters after all items in the list have been printed as described in Section 4.7.4 for the simple PRINT statement. Using this option, you may complete a partially filled line with subsequent PRINT or PRINT USING statements. You may not end a PRINT USING statement with a comma as you can a simple PRINT statement.

BANKUSING is a program which illustrates that output can be arranged in columns so that the decimal points line up normally. Additionally, a dollar sign can be printed immediately before each amount.

```
100 PRINT "ITEM", "  AMOUNT", "  BALANCE"
105 PRINT
110 LET C = 0
120 LET D = 0
130 REM C COUNTS THE NUMBER OF CHECKS
135 REM D COUNTS THE NUMBER OF DEPOSITS
140 READ B
141 REM
142 REM SET UP FORMAT STRINGS IN F$ AND G$
143 LET F$= "<########   $–### . ##     $+#### . ##"
144 LET G$= "<#######                    $–#### . ##"
145 REM
146 REM A SPECIAL FORMAT IS NEEDED FOR THE
147 REM OPENING AND CLOSING BALANCES, WHICH
148 REM HAVE NO TRANSACTIONS
149 REM
150 PRINT USING G$, "OPENING", B
160 REM
170 READ T
180 IF T = 0 THEN 400
190 IF T < 0 THEN 300
200 REM
210 REM HERE FOR A DEPOSIT
220 LET D = D + 1
230 LET B = B + T
240 PRINT USING F$, "DEPOSIT", T, B
250 GOTO 170
260 REM
300 REM HERE FOR A CHECK
310 LET C = C + 1
320 LET B = B + T
330 PRINT PRINT USING F$, "CHECK", –T, B
340 IF B > = 0 THEN 170
350 LET B = B – 1
360 PRINT USING F$, "OVERDRAFT", 1, B
370 GO TO 170
380 REM
400 REM HERE FOR CLOSING
```

```
410 LET S = .03 * D + .06 * C + .60
420 LET B = B – S
430 PRINT USING F$, "SERVICE", S, B
440 PRINT USING G$, "CLOSING", B
470 REM
500 DATA 100.00
510 DATA –23.75, –10.40, 50.00, –7.25, –42.50
520 DATA –45.67, –22.95, 40.00, –50.33, 66.75, 0.00
999 END
```

A run of this program, BANKUSING, is below:

ITEM	AMOUNT	BALANCE
OPENING		$100.00
CHECK	$23.75	$+76.25
CHECK	$10.40	$+65.85
DEPOSIT	$50.00	$+115.85
CHECK	$7.25	$+108.60
CHECK	$42.50	$+66.10
CHECK	$45.67	$+20.43
CHECK	$22.95	$–2.52
OVERDRAFT	$1.00	$–3.52
DEPOSIT	$40.00	$+36.48
CHECK	$50.33	$–13.85
OVERDRAFT	$1.00	$–14.85
DEPOSIT	$66.75	$+51.90
SERVICE	$1.11	$+50.79
CLOSING		$50.79

4.8 *INPUT CONTROL*

There are some occasions when a user wishes to override the normal BASIC input conventions. For example, commas are usually used to separate a fixed number of entries on a line. The following statements allow somewhat greater flexibility.

4.8.1 *The LINPUT Statement*

LINPUT <list of string variables>

If a program calls for data to be entered from the terminal using an INPUT statement and the data consists of strings containing such characters as quotation marks, leading blanks, ampersands, or commas, then the data used in the BASIC computation may not be the ones desired, for BASIC normally treats such characters in special ways. The LINPUT (remember it as a "line-input") statement provides for entering of an arbitrary sequence of characters into a single string. The characters typed may consist of any ASCII characters, other than a carriage return, which terminates the string; the carriage return character is not included in the string. An example of a LINPUT statement appears in the following program, which counts the number of commas in the input string.

```
10 LINPUT A$
20 FOR I = 1 TO LEN (A$)
30 IF SEG$ (A$, I, I) = "," THEN N = N + 1
40 NEXT I
50 PRINT "THERE ARE"; N; "COMMAS IN THIS LINE."
60 END
```

A run of the program follows.

```
?A,B,C,,D,E
THERE ARE 5 COMMAS IN THIS LINE.

READY
```

More than one variable may follow the word LINPUT if the variable names are separated by commas. A new ? appears for each variable in the list. It is also possible to insert strings to be printed between the variables to be input as with the INPUT statement. See Section 2.7.13.

4.8.2 *The MAT INPUT Statement*

MAT INPUT <list of arrays>

The MAT INPUT statement allows the user to enter data when the program does not know how much data will be input. This feature circumvents cumbersome programs such as the following which is designed to perform the simplest task of adding up a few numbers typed in from the terminal.

```
100 LET T = 0
105 INPUT N
110 LET T = T + N
120 IF N <> 0 THEN 105
130 PRINT "THE TOTAL IS"; T
140 END
```

To use such an awkward program, you must type one number and one carriage return in response to each question mark which is printed by the INPUT statement. When a zero is entered, the program assumes that all the numbers have been entered and the total is printed. Besides being time-consuming, intermediate zeros cannot be entered.

The following program using the MAT INPUT statement is much more convenient to use and performs the same function as the previous program.

```
100 DIM A (100)
105 LET T = 0
110 MAT INPUT A
120 FOR I = 1 TO NUM
130 LET T = T + A (I)
140 NEXT I
150 PRINT "THE TOTAL IS"; T
160 END
```

After a question mark has been printed in response to the MAT INPUT statement in line 110, the user may type any number of numbers separated by commas. When the input line is terminated with a carriage return, the first number entered is in A (1), the second is in A (2), and so on. The number of numbers entered is made available by the function NUM. This function has no arguments and will deliver the number of entries until a new MAT INPUT statement is executed.

Zero, one or any number of entries may appear on a line, the only limit being the size of the line. If one wishes to enter more numbers than can be typed on one line, it is possible to continue typing on additional lines. If the last number 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 MAT INPUT statement may also be used to enter strings into a one dimensional array. Rules for enclosing the strings in quotation marks are the same as given in Section 4.6.2, for the INPUT statement with this addition: the last string entered on a line in response to a MAT INPUT statement must be enclosed in quotation marks if its last character is an ampersand (&).

The rules for entering data into a two-dimensional array are somewhat different. See Section 6.6.2 for more information on the MAT INPUT statement.

4.9 *PROGRAM ORGANIZATION STATEMENTS*

When larger BASIC programs are written, they should not be looked upon as a simple series of statements. They should be organized into units analogous to blocks or sections or paragraphs, so that overall action of the program can be managed in terms of "building blocks" of statements. Once these blocks of statements are written and checked, they can be utilized by a programmer who knows only the function they perform, without his having to bother with individual, detailed statements.

BASIC is a language which is designed to be understandable both by machines and by human beings. A program must be understandable to a human being if he is to be able to verify its correctness, improve the technique, change the theoretical basis of the technique, or explain its value to others. Also, when programs are being developed, they do something – not necessarily what is finally desired; all programs do something, even if it is stopping immediately. It must be possible to determine how a program does what it does, even when it is incorrect. English-language comments (or other natural-language comments) can be incorporated into the body of the text of a BASIC program in order to improve its readability and to aid in its interpretation. These comments do not interfere with the operation of the BASIC program.

4.9.1 *The Apostrophe Convention*

A comment may appear on the same line as a BASIC statement if the comment follows the statement and is separated from it by an apostrophe. This is especially useful for explaining the intent of a single BASIC statement when the importance of that statement is not necessarily clear from the BASIC statement alone. A comment may appear on a line by itself if it is preceded by an apostrophe as shown in the following program segment.

```
100 IF ABS (X) < = 1 THEN 130 'PREVENT NEG SQ ROOT IN 130
110 PRINT "ABS (X) IS GREATER THAN 1 IN LINE 100".
115 'AVOID LINE 130 WITH A GO TO STATEMENT
120 GO TO 140
130 LET Y = SQR (1 – X * X)
140 LET Z = Z – Y
```

4.9.2 *More About REM*

As was pointed out in Section 2.7.11, if the first three characters following the line number of a BASIC statement are REM, then any remarks whatsoever may follow on that line. REM statements may be used to convey the function of a block of statements in a program. Knowing the purpose of the BASIC program (or the purpose of each part of it) facilities checking each of the BASIC statements to verify that the program is correct. Well written REM statements greatly increase the value of a BASIC program to other users by making the intent of the programmer known, i.e., what the program as a unit is supposed to do and how different parts of the program work toward this end.

Since REM statements have line numbers, they can be referred to in GO TO statements or other statements which cause a transfer of control such as the ON – GO TO and IF –THEN statements. It is especially appealing to transfer to a REM statement which describes the purpose of a following block of code. The example in Section 4.10.1 illustrates how apostrophe and REM are used to improve the readability of programs.

4.10 *INTERNAL SUBROUTINES*

In BASIC programs, it often happens that similar calculations must be carried out at several places in the computation. We denote a related group of BASIC statements required to carry out such a calculation as a subroutine. It would be tedious and wasteful to have to copy the statements of the subroutine at every place in the entire BASIC program that such a calculation was to be performed. The GOSUB statement provides a way to transfer control to a subroutine. Control returns to the statement following the GOSUB when a RETURN statement is reached in the subroutine. Alternatively, the ON–GOSUB statement allows branching to one of several subroutines and the IF – GOSUB statement allows a conditional subroutine jump. Since an internal subroutine "block" has no definable start, it is the user's responsibility that it is entered through GOSUB statements exclusively.

4.10.1 *The GOSUB and RETURN Statements*

```
GOSUB <line number>
RETURN
```

The GOSUB and RETURN statements are illustrated in the following example where the subroutine in lines 300-410 calculates the greatest common divisor of two numbers X and Y. The program uses this subroutine to calculate the greatest common divisor of three numbers A, B and C, relying on the fact that GCD(A, B, C) = GCD(GCD(A, B),C).

```
110 PRINT "A", "B", "C", "GCD"
120 READ A, B, C
130LET X = A
140 LET Y = B
150 GOSUB 300
160 LET X = G
170 LET Y = C
180 GOSUB 300
190 PRINT A, B, C, G
200 GO TO 120
210 DATA 60, 90, 120
220 DATA 38456, 64872, 98765
230 DATA 32, 384, 72
250 '
300 REM SUBROUTINE TO CALCULATE GCD
305 LET Q = INT(X/Y)
310 LET R = X – Q * Y
320 IF R = 0 THEN 400
330 LET X = Y
340 LET Y = R
350 GO TO 300
400 LET G = Y
410 RETURN 'TO LINE 160 OR 190
420 END
```

When the program is run, X and Y are set equal to A and B. Line 150 contains a GOSUB to line 300. This is the beginning of a calculaton which sets G equal to the greatest common divisor of X and Y. Line 410 is the RETURN statement which returns to 160, the line following the GOSUB. Subsequently, X and Y are given the values of G and C in order to GOSUB to the GCD subroutine once more. Upon return to 190, the line after the second GOSUB, the answers are printed and the process recycles. In operation, the statement

```
180 GOSUB 300
```

records information about the location of the GOSUB before transferring control to line 300. This is done in such a way that a statement like

```
410 RETURN
```

uses the information stored by the GOSUB statement to return control to the statement directly following the GOSUB. Consequently, a subroutine may have many RETURN statements in it, but the first one which is actually encountered causes control to be returned to the main part of the program.

A GOSUB may be executed inside a subroutine to call still another subroutine. In this nested subroutine arrangement, the first RETURN statement to be executed returns control one level to the statement following the most recently executed GOSUB. The next RETURN statement returns control to the statement following the previously executed GOSUB and so on.

4.10.2 *The ON – GOSUB Statement*

ON <expression> GOSUB <list of line numbers>

The ON – GOSUB statement provides a way of transferring control to one of several subroutines. The statement

```
100 ON X–1 GOSUB 700, 800, 900
```

will cause execution of the subroutine beginning in line 700 if the value of X–1 is 1, execution of the subroutine beginning in line 800 if the value of X–1 is 2, and execution of the subroutine beginning in line 900 if the value of X–1 is 3.

The expression "X–1" could have been any arithmetic expression, including a simple variable. The value of this expression must not be less than 1 and not greater than the number of line numbers listed; if so an error message is given. If the value is not an integer, it will be truncated to an integer. When a RETURN statement is encountered, control is returned to the statement following the ON – GOSUB statement.

4.10.3 *The IF – GOSUB Statement*

IF <expression> <relation> <expression> GOSUB <line number>

The IF – GOSUB statement provides a way of transferring control to a subroutine if some specified condition is met. The statement

```
100 IF A$ = "MARRIED" GOSUB 900
```

will transfer control to line 900 if the condition is true.

The condition may be of either a numeric or a string type.

When a RETURN statement is encountered, control is returned to the statement following the IF – GOSUB statement.

4.11 *INTERNAL FUNCTIONS*

BASIC has a number of built-in functions, such as SIN, LOG, SQR, etc. If the user requires an extension to this set of functions, he has the possibility of writing a definition for a new function in BASIC using a DEF statement.

Naming such functions follows the rules of defining variables in BASIC, but the first two letters of the name must be FN. The postfixed letter(s) of the variable name will determine the type of the function; i.e., FNTEXT$ will return a string data element.

Internal functions as opposed to internal subroutines may have arguments as described below. The number and type of arguments must correspond in the definition and the call; if not,a run-time error message is given.

4.11.1 *One Line DEF Statement*

Sometimes a function definition can be written in a single BASIC statement . Suppose an arcsine function is required.

```
100 DEF FNA(X) = ATN(X/SQR(1 – X * X))
110 PRINT FNA (.707)
120 END
```

Line 100 defines the new arcsine function. In the definition of FNA(X), the variable X is not related to any variable of the same name elsewhere in the program. The DEF statement simply defines the function and does not cause any calculation to be carried out; the variable X is called a *dummy argument.* The appearance of FNA in some other place in the BASIC program (this is known as the place where the function is *called)* causes the calculation denoted in the DEF statement to be executed. When the function is called, the value of the argument of the function (.707) in the above example is substituted for the dummy argument throughout the definition of the function. Arguments i the definition and the call are often called *formal parameters*, respectively *actual parameters.*

DEF statements may appear anywhere in a program and may define functions of more than one variable. For example:

```
100 LET D1 = FNR (201.83, 199.01)
110 PRINT D1
120 DEF FNR(X, Y) = SQR (X * X + Y * Y)
130 END
```

When a function of more than one variable is defined, the list of dummy arguments is separated by commas.

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 parenthesis 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(Q 1$, Q2$)
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".

4.12 *RELATIONAL EXPRESSIONS*

A relational expression has the form:

q_1 op q_2

where q_1 and q_2 are arithmetic or string expressions; op is an operator belonging to the following set:

Operator:	Meaning:
=	Equal to
< > or > <	Not equal to
>	Greater than
> = or = >	Greater than or equal to
<	Less than
= < or < =	Less than or equal to
==	Approximately equal to (not strings!)

A relation is true if q_1 and q_2 satisfy the relation specified by op.

A relation is false if q_1 and q_2 do not satisfy the relation specified by op.

Rules:

1. Use a relational operator between two expressions:

   q_1 op q_2

2. It is not permissible to use the form :

   q_1 op q_2 op q_3

   Instead separate two relational expressions with a logical operator .AND. or .OR. in any of the form:

   q_1 op q_2 .AND. q_3 op q_4

   q_1 op q_2 .OR. q_3 op q_4

3. The evaluation of a relation of the form q_1 op q_2 is from left to right.

   The relations q_1 op q_2, q_1 op (q_2), (q_1) op q_2 and (q_1) op (q_2) are equivalent.

   Examples:

```
A > 5.2
RX–X(5)*A < Y
B–C = .5
X(1) >= X(I–1)
I <= 10
A$ <= B$
```

4.13 *LOGICAL EXPRESSIONS*

A logical expression has the general form:

$$0_1 \text{ op } 0_2 \text{ op } 0_3$$

The forms 0_1 are relational expressions; and op is either the logical operator .AND. indicating conjunction or .OR. indicating disjunction.

The logical operator .NOT. indicating negation appears in the form:

$$.\text{NOT. } 0_1$$

The value of a logical expression is either true or false. Logical expressions are used in IF statements.

Rules:

1. The hierarchy of logical operations is:

First	.NOT.
Then	.AND.
Then	.OR.

2. If L_1 and L_2 are logical expressions, then:

   $.\text{NOT.}L_1$
   L_1 .AND. L_2
   L_1 .OR. L_2

   are logical expressions. If L is a logical expression, then (L) and ((L)) are logical expressions.

3. If L_1 and L_2 are logical expressions and op is .AND. or .OR. then L_1 op op L_2 is always illegal.

4. The logical operator .NOT. may appear in combination with .AND. or .OR. only as follows:

   .AND..NOT.
   .OR..NOT.
   .AND.(.NOT....)
   .OR.(.NOT....)

   .NOT. may appear with itself only in the form:

   .NOT.(.NOT.(.NOT. ....

   Other combinations will cause compiler diagnostics.

5. If L_1 and L_2 are logical expressions, the logical operators are defined as follows:

.NOT.L_1	is false only if L_1 is true
L_1 .AND.L_2	is true only if L_1 and L_2 are both true
L_1 .OR.L_2	is false only if L_1 and L_2 are both false

Examples of Logical expressions:

Valid Expressions:

A < 2 .OR. B = 0
A < 2 .AND. B = 0
A < 2 .OR. B = 0 .AND. C = 1
.NOT. A < 2 .AND. B = 0
.NOT. (A < 2 .OR. B = 0)
X > Y .AND..NOT. X > Y + 2
A$ = "MARRIED" .AND. B$ = "WOMAN"
A$ >< B$ .OR. Y < 0

Illegal Expressions:

A < 2 .NOT..OR. B = 0
X + 5 .NOT. <Y

4.14 *OTHER USEFUL STATEMENTS*

4.14.1 *Multiple Statement Line*

More than one statement can appear on a single line if each statement (except the last) is terminated with a colon (:). Thus, only the first statement can have a line number. An error diagnostic is given if a statement cannot appear in a multiple statement line. Statements which logically belong to each other may now be grouped on one line. Multiple statement lines are legal in immediate mode as well.

4.14.2 *The REPEAT Statement and the @ Variable*

REPEAT <expression> [STEP <expression>] :<statement>: . . .<statement>

REPEAT makes it possible to construct a loop of a single line using the multiple statement feature. The REPEAT statement first assigns one to the system variable @ which is later incremented by one if STEP is omitted. The following statements on the line will be repeated while the value of @ (real) is less or equal to the maximum specified. The example below will change the fifth row of a two-dimensional array:

```
10 REPEAT MAX : INPUT ARRAY (5, @)
```

It is, of course, always possible to exchange a REPEAT with a FOR-NEXT loop construction as follows:

```
10 FOR I = 1 TO MAX
20 INPUT ARRAY (5, I)
30 NEXT I
```

4.14.3 *More About IF*

As previously mentioned, IF branches to a line number following THEN, GOTO, or GOSUB if the relational expression turns out to be true. Having described relational and logical expressions as well as multiple statement lines, it is time to introduce a more advanced use of IF:

IF <logical expression> THEN <statement> : . . .<statement>

Dependent upon the logical expression being true or false, the program will execute the statement(s) following THEN or skip to the next line.

Example:

```
10 IF X = Y THEN N = N + 1
20 IF X < Y .AND. A$ = "YES" THEN PRINT "OK" : N = N + 1
30 IF .NOT. A$ > B$. OR. A% < B% THEN GOSUB 500 : GOTO 100
```

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.

4.14.7 *The COMMON Statement*

COMMON [/[<block>]/] <variable> [([<subscript string>])]
[=<length>] , ...

A program may be divided into independently compiled subprograms that use the same data. The COMMON statement reserves storage areas blank or labelled which can be referenced by more than one subprogram written in BASIC, FORTRAN, NPL or MAC assembly.

The common data structure is static which means that size of arrays and strings is fixed as opposed to local arrays and strings in BASIC. A string array consists of ASCII characters rather than string descriptors; for the rest array layout corresponds to that described in Section 4.4.

<block> is an alphanumeric identifier defining the name of the common block. <variable> is a simple variable or array identifier, subscripted or non-subscripted. The identifier may be previously defined in a type declaration statement.

The list may not contain formal parameters. Arrays must be dimensioned in the common statement by a <subscript string> following the array identifier. If an array is dimensioned in both a common statement and a dimensioned statement, a compiler diagnostic results. The subscript(s) must be constant(s). If the subscript string is empty, this array will be equivalent to the next element in the list. The optional <length> is a constant defining the length of string variables. The length may be any even number in the range 2 - 256. Default length is 16.

A block identifier may be a name of one to seven alphanumeric characters or blank. A non-blank name identifies the storage as labelled common; a blank name identifies blank common. If the name is blank, the first two slashes may be omitted. Only one name may be assigned to labelled common, but the name may be specified more than once.

All common storage areas are assigned together in the order of appearance regardless of the line number.

Examples:

```
10 COMMON A, B, C
20 COMMON //X, Y, Z, Q
30 COMMON/BLOCK/F, G(10), X$=4
40 COMMON/BLOCK2/NAMES$(4,4)=10%, AGE(4%)
```

Common Blocks:

The COMMON statement provides the programmer a means of reserving blocks of storage areas that can be referenced by more than one subprogram, the statement reserves both blank and labelled blocks.

If a subprogram does not use all of the locations reserved in a common block, unused variables may be necessary in the COMMON statement to ensure proper correspondence of common areas:

```
Main Program:    10 COMMON/SUM/A, B, C
Subprogram:      10 COMMON/SUM/E, F, G
```

In the above example, only the variables E and G are used in the subprogram. The unused variable F is necessary to space over the area reserved by B.

Rules:

1. COMMON is non-executable and must precede the first executable statement in the program. Any number of COMMON statements may appear in a program unit.

2. Labelled common block identifiers are used only for block identification within the compiler; they may be used elsewhere in the program as other kinds of identifiers.

3. An identifier in one common block may not appear in another common block. If it does, the identifier is doubly defined and an error message will result.

4. The order of the arrays in a common block is determined by the COMMON statement. No array bound checking is performed.

5. At the beginning of program execution, the contents of the common blocks are undefined. Common variables are assigned values through the LET, INPUT and DATA statements.

6. Common arrays in mat input/output or mat arithmetic statements are not allowed.

7. Common strings are left justified with trailing spaces if necessary.

8. No bound checking when accessing common arrays.

The length of a common block in computer words is determined from the number and type of the list identifiers. In the following statement, the length of the common block A is 26 computer words. The origin of the common block is Q(0).

Examples:

1. Labelled Common

```
10 INTEGER NR
20 COMMON/A/ Q(3), R(3), NR(1)
```

ORIGIN		
	Q	(0)
	Q	(1)
	Q	(2)
	Q	(3)
+12	R	(0)
	R	(1)
	R	(2)
	R	(3)
+24	NR	(0)
	NR	(1)

Each real variable requires three computer words.

2. Blank Common

```
10 INTEGER K, N, M, DUMMY
20 COMMON DUMMY( ), A, B(1), K
30 COMMON N(1), M(1), A$(1)=6%
```

ORIGIN		
	A /DUMMY	(0)
+3	B	(0)
+6	B	(1)
+9	K	
+10	N	(0)
+11	N	(1)
+12	M	(0)
+13	M	(1)
+14	A$	(0)
+17	A$	(1)

ASCII string
6 characters = 3 words

Note that element K may be accessed by DUMMY (9) because no array bound checking is performed.

3. Rearrangement of Common

Main Program: 10 COMMON/EX/TEMP(19)
The labelled common, EX, occupies 60 storage locations.

Subprogram:
```
10 INTEGER I, J
20 COMMON/EX/B(9), I(9), J(19)
```

The labelled common occupies the same 60 storage locations as in the main program. However, 30 locations are used by the real array B, 10 locations are used by the integer I and 20 locations are used by the integer array J.

4.14.8 *The CHAIN Statement*

CHAIN <string expression>

An elementary, but successful, method of dividing a lengthy basic program into manageable segments is to run several programs successfully by typing the necessary OLD/LOAD and RUN commands. In this mode of operation, the user determines the program to be executed next simply by typing the proper name after the OLD/LOAD command.

An automatic way of running another program is to use a CHAIN statement. The word CHAIN is followed by < string expression> forming the file name of the next program. Chained program units must be previously compiled to BRF. The CHAIN statement is the last statement executed in each program segment other than the last segment.

CHAIN implies automatic loading (LOAD) and starting (RUN) of a program. In fact, the statement will act as a command because it may execute in immediate mode. The string expression may optionally contain several file names (or numbers) separated by commas or spaces. The syntax corresponds to that of the LOAD command.

Chaining to precompiled (BRF) program units is considerably more efficient than chaining to BASIC source which would require compilation upon each call.

When a CHAIN statement is encountered the running of the current program is terminated and execution of the designated program begins. This procedure requires the BASIC compiler in memory; thus, an error message will appear if the loading is done by another BRF loader.

Values of local variables in one program are not passed unchanged to a subsequent program, but are always set to zero at the beginning of each program execution, unless the variables are declared in a COMMON statement (see Section 4.14.7). When using common variables for parameters, the data remains in the main high speed memory of the computer.

Communication between chained programs may, of course, be performed by means of files, but this involves a physical transfer of data to/from an external storage device.

A program is usually segmented by using subprograms rather than by chaining if the user wishes to preserve variable values between segments or if phases in a program re-occur. The chaining technique is sometimes necessary when the subprogram technique fails to reduce the program enough so that it will execute in the computer memory allotted. When a program uses subprograms, space required is determined by the main program and the largest subprogram. When the chaining technique is used, only enough memory for the successful run of the largest program is required. However, each call to a subprogram different from the last one called requires a physical transfer from an external storage device. This entails a considerable amount of time, and applications will only be practically successful if they call for new subprograms a limited number of times.

The following program gives the user the option of playing one of three games. The number input by the user corresponds to the location in A$ which contains a string consisting of the corresponding game. In line 150, a chain is made to the program having this file name:

```
100 REPEAT 3: READ A$(@)
110 DATA BONDESJAKK, LUNAR-LANDER, POKER
120 PRINT "TYPE 1 TO PLAY BONDESJAKK, 2 TO PLAY"
130 PRINT "LUNAR-LANDER, AND 3 TO PLAY POKER."
140 INPUT I
150 CHAIN A$(I)
200 END
```

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 calls.

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
```

5.3 *RANDOM ACCESS FILES AND VIRTUAL ARRAYS*

The major use of random access files is to hold big amounts of data which should be accessed in a random manner. The data will normally be loaded from a sequential file using a BASIC program, or be generated by a program.

Random files are used to hold numbers and strings. The data is manipulated internally in BASIC and accordingly the internal format is used. Numbers are represented in the standard floating point or integer formats and strings are saved in ASCII code two characters to a word.

The addressing mode of arrays is used to address the individual items in a random access file and when an array is assigned to a random access file, the associated indexed variable may be used the same way as for standard arrays. Such arrays are called virtual arrays or matrices.

The array format is used to access data because the PRINT and INPUT statements deal only with the next sequential data element. A sequential file, then, is limited in its applications and depends on a strictly sequential treatment. With virtual arrays, the user can reference any element of one or more matrices within the file, independent of where in the file that element resides. This random access of data allows the user (non-sequential) referencing of the data for use in BASIC. The virtual arrays are read into memory automatically by the system.

Data stored in virtual arrays remain, even after the terminal is logged out. The data can be retrieved later by accessing that file from BASIC or other program systems. It is illegal to use MAT statements with virtual arrays. Note also that there is no bound checking when accessing arrays.

5.3.1 *Opening a Random Access File*

Before any virtual array access is made, the random file must be opened, associating the file name with a connect device identifier. The identifier must always have a numeric value in the range 0-127 (which is also used in the virtual DIM statement). The access mode is RANDOM allowing read as well as write random access. It is very important to close random files, because the CLOSE statement causes output of the last block.

Example:

```
100 OPEN # 50: FOR RANDOM "MYFIL"
900 CLOSE # 50:
```

5.3.2 *Declaring Virtual Arrays (Virtual DIM Statement)*

The BASIC program has to be informed that a particular array is not to be stored in the memory, but on a random file. This is declared in a special form of the DIM statement:

DIM # <expression>:<list>

The expression denotes the connect device identifier which is the same referred to in the OPEN statement. The list must appear as it would in a standard DIM statement. (See Section 2.7.7.) Subscripted variables of any type can be virtual array elements. More than one array can be specified within one random file.

Remember that future references should always dimension the arrays to the same size. There are no restrictions on the array size, because the subscripts are converted to and computed in double integer format.

```
20  DIM # 3:A(100,100),A$(100%,100%),I%(2000%),DBI%%(1000)
```

The above statement indicates that the file associated with # 3 contains 10201 real numbers addressed by:

A(0,0),A(1,0),A(2,0)...,A(100,0),A(0,1),....etc.

Then follow 10201 strings. The maximum number of characters in each is 16, because no size is given. The strings are addressed by:

A$(0,0),A$(1,0)..., A$(100,0),A$(0,1),....etc.

Thereafter follow 2001 integer numbers and 1001 double integer numbers.

5.3.3 *Virtual String Arrays*

Standard strings are of variable length, from 0 to 32767 characters. Virtual string array elements are of fixed length, from 2 to 512 characters which is the maximum length. If no length is specified, a default length of 16 characters is assumed. The fixed length can be changed by the program, but a syntax error will be printed if you don't follow this rule: (length + 1)/2 must be a multiple of 512. The total space is always reserved for each element, but an element need not use the maximum length, The maximum length is optionally specified in the virtual DIM statement and must always be a constant.

```
10  DIM # 5: A$(10000) = 1,B$(100,100),C$(100) = 32%
```

The statement above reserves space on the file associated with # 5 for:

A$:	10001 strings of maximum 2 characters each
B$:	10201 strings of maximum 16 characters each
C$:	101 strings of maximum 32 characters each.

5.3.4 *Using a Random Access File From a Program*

Our example from sequential files which computes the average grade for each of several students in a group is now used again in a new version. — Suppose we want to print out the average grade for a given student, i.e. the student name is the key for further computations. It is obvious that we could solve the problem by using sequential files, but we soon realize that using arrays are much more convenient and faster. Suppose also that the number of students has increased so that the total amount of data is too big to be held in arrays in memory.

Before running the program the file must be initialized. The array NAMES$ is filled with the student names, and the two dimensional array GRADES is filled with the corresponding grades. The following program, AVERAGE4, is used to scan the data base and perform the necessary computations and printouts:

```
100  REM PROGRAM NAME -- AVERAGE4
110  '
120  REM THIS PROGRAM COMPUTES AVERAGE GRADES FOR A GIVEN STUDENT.
130  REM THE NAMES AND GRADES ARE STORED IN A RANDOM ACCESS FILE
140  REM CALLED STUDENT-FILE.
150  REM THE NAMES ARE FOUND IN THE ARRAY NAMES$.
160  REM THE GRADES ARE FOUND IN THE ARRAY GRADES.
170  REM THE INDEX BY WHICH THE NAME IS STORED IS THE STUDENT
180  REM NUMBER.  THIS NUMBER INDICATES WHERE IN THE ARRAY
190  REM GRADES TO FIND THE GRADES.
200  REM
210  OPEN # 1: FOR RANDOM "STUDENT-F:DATA"
220  DIM # 1: NAMES$(1000)=32%, GRADES(4,1000%)
230  PRINT !! : LINPUT "STUDENT NAME", STUD$
240  IF STUD$ = "FINISHED" GOTO 320
250  PRINT ! : AVER=0
260  REPEAT 1000: IF NAMES$(@)=STUD$ THEN I%=@ : GOTO 280
270  PRINT "NO SUCH STUDENT" : GOTO 230
280  REPEAT 4: AVER=AVER+GRADES(@,I%)
290  AVER=AVER/4
```

```
300  PRINT "NO.";I%,NAMES$(I%), "AVERAGE";AVER
310  GOTO 230
320  CLOSE # 1 :
330  END
```

Let us go through the program and make some observations before starting the execution. The first executable statement is located in line 210 opening the file really named: STUDENT-FILE. We see that file names may be abbreviated until ambiguity arises. The word DATA following the colon is the file type, and has to be specified unless the file is of type SYMB, which is the default type in OPEN statements. — The next statement declares the names and sizes of the virtual arrays. Space is reserved for a maximum of 32 characters for each string element in NAMES$. GRADES is a real array.

Note that the indexes (subscripts, dimensions) may be of any type. In line 230 we find a multiple statement line, first two blank lines are given, and then the program will ask for the student name. This is an example showing the convenience of *printing messages* in an input statement. LINPUT is used to avoid typing of ". The next statement will close the file and stop execution if the student name, FINISHED is typed. Line 250 gives a blank line and initializes AVER for computation of the average.

The REPEAT statement in line 260 initializes a loop of maximum 1000 to find the given student in the data base. If found, the @ variable (real) indicates the student number which is assigned to the integer I% before going on to line 280. If not found @ takes the value 1001, a message is printed in line 270 and a new student name is asked for. — In line 280 we now realize the importance of storing the previous value of @, because this variable is assigned to 1 in the following REPEAT statement.

The grades are found in the I%'th column and are added by letting the row (@) run from one to four. It is important to notice that arrays are stored by columns when working with virtual arrays! In this example we obtain the effect of accessing four sequential elements. Line 290 divides the sum of the grades by the number of grades, and line 300 prints the result. The unconditional GOTO statement in line 310 jumps back to ask for a new student name.

The following is a run of AVERAGE4:

```
STUDENT NAME?ADA SHAW

NO. 876      ADA SHAW          AVERAGE 74.75

STUDENT NAME?GERALD FRIENT

NO SUCH STUDENT

STUDENT NAME?GERALD FRIEND

NO. 54       GERALD FRIEND     AVERAGE 78.75

STUDENT NAME?PHILIP CLOUGH

NO. 318      PHILIP CLOUGH     AVERAGE 83

STUDENT NAME?PER PEDERSEN

NO SUCH STUDENT

STUDENT NAME?FINISHED

READY
```

6 ARRAY MANIPULATIONS

6.1 *INTRODUCTION*

Up to this point in the manual a singly subscripted variable (a variable having only one subscript) has denoted a one-dimensional array and a doubly subscripted variable (a variable having two subscripts) has denoted a two-dimensional array. In this chapter it is appropriate to refer to vectors and matrices since we are describing them in a mathematical context.

Vectors and matrices are both *arrays.* That is, an array is denoted by a variable having one or more subscripts; a matrix is an array having two subscripts.

A *string array* is an array whose entries are strings.

BASIC provides MAT statements which are designed to allow the programmer to work with arrays in a simple and straightforward manner. Although arrays have a row number 0 and a column number 0 in BASIC (Sections 2.4 and 4.4), the MAT statements generally ignore them. Virtual arrays cannot be used with MAT statements. Double integer arrays are allowed only with MAT input/output statements. The type of the arrays involved in a MAT arithmetic operation must always correspond, i.e. *mixed mode is not permitted.*

6.2 *MAT INITIALIZATION STATEMENTS*

There are three MAT statements which facilitate the procedure of assigning values to individual array entries.

```
100  MAT A% = ZER
```

This statement assigns a value of zero to each entry of the integer array A%.

```
110  MAT A = CON
```

This statement assigns a value of one to each entry of the array A.

```
120  MAT A = IDN
```

This statement sets the matrix A equal to the identity matrix. For this statement to be valid A must be a square matrix: A must be doubly subscripted and have its number of rows equal to its number of columns. A may not be a vector.

All three of these MAT statements do not affect row 0 or column 0 of the arrays on which they operate.

6.3 *CHANGING DIMENSIONS USING MAT STATEMENTS*

As described in Sections 2.4 and 4.4 the DIM statement is used to dimension (i.e., to reserve space in the computer for) subscripted variables. Space for entries in row 0 and column 0 of an array is a part of the total space reserved. For example the statement

```
100  DIM A(7), B(11,5)
```

results in 8 spaces being reserved for A with room for entries 0 through 7. (11+1)*(5+1) = 72 spaces are reserved for B with room for entries in rows 0 through 11 and columns 0 through 5. If subscripted variables are used in a program but do not appear in a DIM statement, BASIC implicitly saves 11 spaces for a vector and 121 spaces for a matrix (a maximum of 10 for each subscript).

It is possible to change the dimensions of the arrays used in some MAT statements by specifying the desired dimensions in the statements themselves. The initialization statements allow this flexibility. The statements

```
100  DIM A(8)
110  MAT A = ZER(5)
```

will reserve nine spaces for the vector A in line 100 and A will be redimensioned (that is, the space reserved for A in the computer will change) to a vector having 6 entries (entries 0 through 5) in line 110 with A(1) through A(5) set equal to zero. A reference to A(6) after line 110 will cause an error message to be output and the program run will terminate.

Redimensioning variables in the MAT statements may cause dimensions which exceed the space previously reserved for the arrays. In the previous example we may retype line 110 to read

```
110 MAT A = ZER(15)
```

Matrices may also be redimensioned in the MAT... CON statement.

```
100  DIM M(8,2)
110  MAT M = CON(5,3)
```

Twenty-seven spaces are stored for M in line 100 and line 110 requires 6*4 = 24 spaces for the redimensioning of M. Again, the space required for redimensioning may exceed the spaces reserved.

Matrices may be redimensioned by using the MAT ... IDN statement and the desired number of rows and columns is included in parentheses as in the preceding examples.

```
100  DIM A(6,5)
110  MAT A = IDN(6,6)
120  END
```

Here the matrix A is dimensioned to be 6 by 5 and in line 110 it is set equal to the 6 by 6 identity matrix.

A vector may be redimensioned to a matrix or vice versa.

As with subscripts, dimensions designated in MAT statements do not have to be integers: any arithmetic expression may be used, and if the value of the expression is not a whole number, its integer part is used.

Redimensioning of arrays may occur in other MAT statements. This feature will be noted as the remaining MAT statements are discussed.

6.4 *ARITHMETIC OPERATIONS*

```
110  MAT C = A+B
120  MAT C = A–B
```

The first statement causes the array C to be the sum of the two arrays A and B. In the second statement C is the result of subtracting array B from array A. A and B may be vectors or matrices as long as they both have the same dimensions.

The array C is *redimensioned* if not matching A or B.

```
100  MAT A = B
```

This statement sets each entry of the array A equal to the corresponding entry of B. A is *redimensioned* if not matching B.

```
130  MAT C = A*B
```

This statement puts the product of arrays A and B into array C, which is *redimensioned* if not matching. If A and B are matrices (that is, they have two subscripts), the number of columns in A must be equal to the number of rows in B. C will have the same number of rows as A and the same number of columns as B; thus, if A is an M by N matrix and B is an N by P matrix, then C will be a M by P matrix.

Vectors may be used in matrix multiplication. If a vector A is multiplied with a matrix B in a statement MAT C = A*B, then A must have the same number of entries as B has rows. The product is a vector with the same number of entries as B has columns; thus, if A is a vector with N elements and B is a matrix with N rows and P columns, then C will be a vector with P elements. If A is a matrix and B is a vector, they can again be multiplied together in the statement MAT C = A*B. This time, the vector B must have the same number of elements as the matrix A has columns; C will be a vector with the same number of elements as A has rows. Thus, if A has M rows and N columns and B has N columns, the resulting vector C will have M elements.

If two vectors are multiplied together to produce what is somtimes called the dot or inner product, the result will be a single number. The two vectors being multiplied must have the same number of elements. Thus, in the statement MAT C = A*B, if A and B are vectors, they must have the same number of entries. The product will now be put into C(1) or C(1,1), but no redimensioning will take place.

While the statements

```
100  MAT A = A+B
110  MAT A = A–B
```

are allowed, the statement

```
120  MAT A = A*B
```

will result in an error message. When adding or subtracting two arrays, any entry of the array is only used once so that the answer may be stored immediately in the array. If entries of the matrix being operated on during a multiplication are replaced, components needed to complete the matrix multiplication are destroyed.

The following matrix multiplication is valid, provided A is a square matrix.

```
100  MAT C = A*A
```

Performing more than one arithmetic operation in a single MAT statement is illegal. Thus, to evaluate the expression A+B–C two MAT statements are required. One way of evaluating the expression follows. We assume all dimensions are correct.

```
100  MAT D = A+B
110  MAT E = D–C
```

In general these MAT statements ignore row 0 and column 0 of the arrays on which they operate.

6.5 *FUNCTIONS*

The transpose of a matrix may be found using the following statement:

```
100  MAT C = TRN(A)
```

This statement sets matrix C equal to the transposed version of A. If A has N rows and P columns, C will be *redimensioned* to have P rows and N columns if necessary. The statement

```
110  MAT A = TRN(A)
```

is illegal.

The statement (called scalar multiplication)

```
100  MAT C = (K)*A
```

causes each entry of array A to be multiplied by the value of K to form the corresponding entry of the array C which is *redimensioned* to be the same size as A if necessary. K may be any constant, variable name or arithmetic expression and must be enclosed in parentheses. The statement

```
100  MAT A = (K)*A
```

is legal.

The statement

```
100  MAT C = INV (A)
```

sets matrix C equal to the inverse of matrix A. A must be a square matrix, and C is *redimensioned* to be the same size as A if necessary. Matrix inversion can involve arrays of real type only.

The function DET is available *after* an inversion is performed, and it is the value of the determinant of the matrix whose inverse was computed. It is important to point out that even though a matrix whose determinant is zero has no inverse, trying to compute the inverse of such a matrix in the above MAT statement will not cause the program run to stop or cause the output of any kind of error message. In this case DET is set equal to zero and the resulting "inverse" matrix is obviously *not* correct. It is up to the user to check the value of DET to determine whether or not the matrix has an inverse.

Since DET is not available until after the inverse is found, if the value of the determinant of a matrix is desired the inverse of the matrix must be computed first.

The following statement is legal:

```
100  MAT A = INV (A)
```

All three of these functions may change the values stored in row 0 and column 0 of the arrays involved. When inversion takes place, row 0 and column 0 of the inverse matrix are used to store intermediate calculations.

6.6 *INPUT AND OUTPUT OPERATIONS*

6.6.1 *The MAT READ, MAT PRINT and MAT PRINT USING Statements*

There are MAT statements that cause entire arrays to be input or output. The program MATRIX

```
100  DIM M(3,5)
110  MAT READ M
120  DATA 1, 2, 3, 4, 5, 6, 7, 8, 9
130  DATA 10, 11, 12, 13, 14, 15
140  END
```

will cause fifteen numbers to be read into the matrix M by rows. That is, the first row of M is read in, then the second and finally the third. Row 0 and column 0 are not affected. If the following line is added to MATRIX

```
135  MAT PRINT M
```

and line 110 is retyped as

```
110  MAT READ M(2,6)
```

the program will yield the following output when it is run:

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

M is redimensioned in line 110 to be a two by six matrix. Twelve numbers are read into M. Line 135 causes M to be printed in matrix format: the entries of each row are spaced five to a line and each row begins on a new line. Row 0 and column 0 are not printed, and a blank line is output before the first row of the matrix is printed.

If line 135 of MATRIX is changed to read

```
135  MAT PRINT M;
```

the following output is produced when MATRIX is run:

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

The semicolon after the matrix name causes M to be printed with the entries of each row closely packed on a line.

The MAT READ and MAT PRINT statements may be used with vectors as well as with matrices. The format of the statements is that described for matrices. The program VECTOR

```
100  DIM V(3)
105  MAT V = CON
110  MAT PRINT V
120  END
```

will cause V to be printed as a column of numbers:

```
VECTOR
1
1
1
READY
```

If line 110 of VECTOR is changed to read

```
110  MAT PRINT V,
```

the entries of vector V are spaced five numbers to a line in row format as follows:

```
VECTOR
1       1       1
READY
```

If a semicolon replaces the comma in the new line 110, V is printed in row format with the entries of V closely packed.

More than one array name may appear in a single MAT READ or MAT PRINT statement. In the MAT PRINT statement commas and semicolons are used both to delimit the names and to control the format in which the arrays are printed. For example, in the statement

```
100  MAT PRINT V, M;
```

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.

6.7 *EXAMPLES USING MAT STATEMENTS*

The following two examples illustrate some of the MAT statements discussed in this chapter.

6.7.1 *MAT Arithmetic*

```
100  READ N,P
110  MAT READ A(N,N)
120  MAT B = CON(N,N)
130  MAT C = A+B
140  PRINT "SUM OF A AND MATRIX OF 1'S IS"
150  MAT PRINT C
160  PRINT
170  PRINT "INPUT"; N*P; "VALUES FOR MATRIX B";
180  MAT INPUT B(N,P)
190  MAT C = A*B
200  PRINT
210  PRINT "PRODUCT OF A AND B IS"
220  MAT PRINT C;
230  MAT D = TRN (C)
240  PRINT
250  PRINT "TRANSPOSE OF THIS PRODUCT IS"
260  MAT PRINT D
270  DATA 2,3
280  DATA 1,2,3,4
290  END
```

Since the matrices used in this example do not appear in a DIM statement, BASIC implicitly dimensions them to be ten by ten and reserves 121 spaces for each matrix. Line 110 dimensions A to be 2 by 2, while it reads values for the entries of A from the DATA statement in line 280. Line 120 dimensions B to be 2 by 2 and sets all entries of B equal to 1. Line 130 adds A and B and stores the result in C. C is redimensioned to be a 2 by 2 matrix as is shown when it is printed in line 150. Line 180 requests the user to input enough values to fill a 2 by 3 matrix and B takes on these new dimensions. Line 190 sets C equal to the product of A and B and C is redimensioned to be 2 by 3. C is printed in closely packed format in line 220. Matrix D becomes the transpose of C in line 230 and D is redimensioned to 3 by 2. D is printed in regular format in line 260.

A run of this example follows:

```
EXAMPLE1

SUM OF A AND MATRIX OF 1'S IS

 2             3
 4             5

INPUT 6 VALUES FOR MATRIX B? 2, -1, 7
?18, 6, -10

PRODUCT OF A AND B IS

 38      11      -13
 78      21      -19

TRANSPOSE OF THIS PRODUCT IS

 38              78
 11              21
-13             -19

READY
```

6.7.2 *Inverting a Matrix*

The second example inverts an N by N Hilbert matrix which has the form

1	1/2	1/3	...	1/N
1/2	1/3	1/4	...	1/(N+1)
.	.	.	...	
.	.	.	...	
.	.	.	...	
1/N	1/(N+1)	1/(N+2)	...	1/(2N–1)

A listing of the program follows:

```
100  REM THIS PROGRAM INVERTS AN N BY N HILBERT MATRIX
110  DIM A(20,20), I(20,20), B(20,20)
120  DIM C(20,20), D(20,20)
130  READ N
140  MAT A = CON(N,N)
150  FOR I = 1 TO N
160  FOR J = 1 TO N
170  LET A(I,J) = 1/(I+J-1)
180  NEXT J
190  NEXT I
```

```
200  MAT B = INV(A)
210  PRINT "INV(A) ="
220  MAT PRINT B;
230  PRINT
240  PRINT "DETERMINANT OF A =";DET
260  MAT I = IDN (N,N)
270  MAT C = A*B
280  MAT D = I-C
290  FOR I = 1 TO N
300  FOR J = 1 TO N
310  IF X>=ABS(D(I,J)) THEN 330
320  LET X = ABS(D(I,J))
330  NEXT J
340  NEXT I
350  PRINT
360  PRINT "LARGEST ABSOLUTE DIFFERENCE ="; X
370  DATA 4
380  END
```

The double loop in lines 150 - 190 sets up the Hilbert matrix A after the correct dimensions have been set up in line 140. A single instruction results in the computation of the inverse (line 200) and one more instruction prints it out in closely packed format (line 220). The value of the determinant of A is available after the inversion and is printed in line 240. I is set equal to the identity matrix having N rows and N columns in line 260. Lines 270 through 340 find the largest absolute difference between an entry of the product matrix A*B and the corresponding entry of the identity matrix. This value is printed in line 360 and is a measure of the accuracy of the inverse since the product of a matrix and its inverse is the identity matrix.

The following run uses a value of 4 for N.

```
HILMAT
INV(A)=
  16      -120       240      -140
-120       1200     -2700      1680
 240      -2700      6480     -4200
-140       1680     -4200      2800

DETERMINANT OF A = 1.65344 E -07
LARGEST ABSOLUTE DIFFERENCE = 1.66893 E -06
READY
```

While this example shows how several MAT statements are used, it also points out that the accuracy of the matrices generated by using MAT statements depends on the structure of the matrices and on the fact that the computer stores any number to only a limited number of significant digits. These two factors combine in this example when N is greater than or equal to 7 to cause severe roundoff errors which in turn cause a highly inaccurate inverse to be returned. When N = 7, a value for the absolute difference described previously is greater than one and continues to grow as N increases.

6.8 *SIMULATING AN N–DIMENSIONAL ARRAY*

Although arrays having more than two dimensions are not allowed in BASIC, the method outlined in the following program can be used to simulate an array having any number of dimensions. It makes use of the fact that defined functions may have any number of arguments, and a one to one correspondence is set up between the entries of the array and the entries of a vector. Formatting techniques cause the entries of the vector to be printed in a format reflecting the dimensions of the array.

This example simulates an array having three dimensions; it can easily be rewritten to accomodate four or more dimensions.

```
100  DIM V(1000)
110  MAT READ D(3)
120  DEF FNA(I,J,K) = ((I–1)*D(2)+(J–1))*D(3)+K
130  FOR I = 1 TO D(1)
140  FOR J = 1 TO D(2)
150  FOR K = 1 TO D(3)
160  LET V(FNA(I,J,K)) = I+2*J+K↑2
170  PRINT V(FNA(I,J,K)),
180  NEXT K
190  PRINT
200  NEXT J
210  PRINT
230  NEXT I
240  DATA 2,3,4
250  END
```

When the program is run, the vector is printed as two 3 by 4 matrices.

```
3–ARRAY

 4        7        12        19
 6        9        14        21
 8       11        16        23

 5        8        13        20
 7       10        15        22
 9       12        17        24

DONE
```

6.9 *THE ROW ZERO AND COLUMN ZERO*

The zeroth row and column of a matrix can be used to store information, provided that no MAT operation is intended to affect it. In very large programs which do not use the MAT operations, this fact can be used to good advantage. An array must be dimensioned to be 200 by 10 to store 2000 items of information if the zeroth row and column are not used; BASIC sets aside 2211 places for that array. If the zeroth row and column were used, the dimensions could be set to be 199 by 9, and only 2000 places would be reserved, The program would be smaller and might be able to run in the space allotted when without this redimensioning the program would occupy too much storage to run.

7 PROGRAMS, FUNCTIONS AND SUBPROGRAMS

7.1 *PROGRAM UNITS*

The words function, subroutine or subprogram within this chapter refer to *external functions or subroutines* as opposed to internal functions or subroutines (FNA(X), GOSUB 10).

A NORD BASIC program consists of one main program and, optionally, one or more subprograms. The term program unit refers to either the main program or a subprogram.

A main program is a set of statements and comments forming a self-contained computing procedure; it must contain at least one executable statement. A PROGRAM statement may be used as the first statement of a main program, but is not necessary. A main program may not contain a FUNCTION, or a SUBROUTINE statement.

A subprogram is also a set of statements and comments, but is headed by either a FUNCTION or SUBROUTINE statement.

All program units must be terminated by an END statement. The main program and subprograms communicate with each other through parameters, virtual arrays, or sequential files.

7.2 *MAIN PROGRAM*

A main program may be written with or without references to subprograms.

The PROGRAM statement may be used as the first statement of the main program, and has the following format:

PROGRAM <name>

name is an alphanumeric identifier from one to six characters; the first must be alphabetic. This name must not be mixed up with the leader printed with RUNH and LISTH. The statement is optional.

A main program may refer to both subroutines and functions which are compiled independently of the main program. A calling program is a main program or subprogram that refers to subroutines and functions.

7.3 *PARAMETERS*

Main programs, subprograms, and functions use parameters as one means of communication. The parameters appearing in a subroutine call or a function reference are *actual* parameters. The corresponding parameters appearing with the subroutine or function name in the definition are *formal* parameters. Actual and formal parameters must agree in order, type and number.

7.3.1 *Formal Parameters*

The following are permissible forms for formal parameters:

array name
simple variable

Since formal parameters are local to the subprogram containing them, they may be the same as names appearing outside the program unit.

No element of a formal parameter list may appear in an EXTERNAL or CALL statement within the subprogram. When a formal parameter represents an array, it should be declared in a DIM statement within the subprogram.

Example:

```
10 SUBROUTINE PER(A,I,X)
10 FUNCTION OLE(X)
```

A, I and X are formal parameters.

7.3.2 *Actual Parameters*

The following are permissible forms for actual parameters:

constant
simple variable or matrix element
arithmetic expression
array name
program name

When an actual parameter is a program name, that name must also appear in an EXTERNAL statement (especially meant for RT applications).

7.4 *FUNCTION SUBPROGRAM*

A function subprogram is a computational procedure which returns a single value associated with the function name. The type of the function is determined by its name in the same way as a variable identifier.

The first statement of a function subprogram must have the following form:

FUNCTION <name>(<formal parameter list>)

The name of the function must also appear as a variable name in the defining subprogram. The value of this variable at the time of execution of the END statement in this subprogram is called the value of the function.

The function subprogram may contain any statement except SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined. Recursive calls are not permitted.

Integer/double integer functions may be declared by mentioning the function name in a type declaration statement.

7.4.1 *The EXTERNAL Statement and Function Reference*

If an external function is to be referenced its name must be declared in an EXTERNAL statement:

EXTERNAL <name 1>,<name 2>,. . . .

where name 1, name 2.... are the referenced function names. The EXTERNAL statement must be entered into the BASIC compiler before the function references.

A function is referenced by

<name>(<actual parameter list>)

where name identifies the function being referenced. It is the same as the name in the FUNCTION statement.

A function reference may appear any place in an expression where an operand may be used. The evaluated function will have a single value associated with the function name. When a function reference is encountered in an expression, control is transferred to the function indicated. When the END statement in the function subprogram is encountered, control is returned to the statement containing the function, with the function reference replaced by the value of the function.

Example:

```
10 EXTERNAL XSQ
20 X = A + B * XSQ(D)
```

7.4.2 *Function Parameters*

When a function reference is executed, actual parameters are associated with all appearances of the corresponding formal parameters in executable statements in the defining subprogram. If a formal parameter appears in a statement redefining its value, the corresponding actual parameter must be a simple variable or a matrix element. If an actual parameter is an arithmetic expression, it is evaluated and its value is associated with the corresponding formal parameter.

If a formal parameter is a matrix name, the corresponding actual parameter must be a matrix.

A function need not have any parameters, but maximum 63 are permitted.

7.5 *SUBROUTINE SUBPROGRAMS*

A subroutine is a computational procedure which may return none, one, or more values. No value or type is associated with the name of a subroutine. The first statement of a subroutine subprogram must be the following:

SUBROUTINE <name>[(<formal parameter list>)]

where name is an alphanumeric identifier as in the PROGRAM statement. The formal parameters may be variable names, array names, or subprogram names.

The name of the subroutine must not appear in any other statement in the subprogram. The parameters may be defined or redefined within the subprogram so that they may effectively return results. The same rules apply to the parameters as for function subprograms.

No value is associated with the name of the subroutine, and the subroutine must be referenced by a CALL statement.

7.5.1 *The CALL Statement*

The executable statement in the calling program to refer to a subroutine is the form:

CALL <name>[(<actual parameter list>)]

The name may not appear in any specification statement in the calling program except in EXTERNAL statement. A subroutine may also be referenced by the appearance of its name in an EXTERNAL statement.

The CALL statement transfers control to the subroutine. When the END statement is encountered in the subroutine, control is returned to the next executable statement following the CALL in the calling program.

Examples:

1) *Subroutine Subprogram*

```
 10  SUBROUTINE PIP(A,B,C)
 20  A = B**C
      .
      .
100  END
```

2) *Calling Program Reference*

```
      .
      .
      .
30  CALL PIP(V(1),X,3)
40  REM PARAMETERS MUST AGREE IN NUMBER AND TYPE
```

7.6 *COMPILATION AND EXECUTION WITH SUBPROGRAMS*

Within the BASIC system only one of the present program units may exist as an incremental unit. Such a unit has the following characteristics:

- The statements may be changed.
- Its identifiers may be examined and changed.
- The run-time error messages reference the line number where the error occurred.
- Break-points may be set.

All other units may be regarded as static blocks where changes must be made by editing and compiling into the binary relocatable format (BRF).

The transformation into BRF format is obtained by using the command

COMPILE <source-file>[<list-file><BRF object-file>]

which starts a compilation of the program unit(s) in the source file.

If the first parameter is present only, this command acts like OLD except for two deviations:

- The new program is appended to any old one with no system initiation.
- File-name will not be taken as the program name.

If the second parameter is present, a listing of the program will be obtained on the list-file/device specified.

When the third parameter is missing, this indicates that the compilation is done incrementally which is the normal use of the system. However, if the third parameter is present, the compiler will translate the source-file program unit(s) into BRF format which is written on the file/device specified. Default file type is :BRF. Such source-files should be terminated with an EOF statement:

<line number> EOF

7.7 *MAIN PROGRAM AND SUBPROGRAM LINKAGE*

Subprogram units that are referenced in the main program must be entered into the system by the command

LOAD <BRF object-file>

One or more object-files may be specified delimited by spaces.

When a load is completed the current load address and the memory upper bound is printed on your terminal in the format

FREE: <current location>–<upper bound>

Now a list of the loaded units and their memory addresses may be obtained by typing

ENTRIES-DEFINED [<file-name>]

Any referenced but still undefined entry may be examined by

ENTRIES-UNDEFINED [<file-name>]

When all referenced entry-points (units) are present you may start your program, else you get a message which tells you the undefined entries.

Subprograms on BRF object files may have been created from NORD STANDARD FORTRAN, NORD BASIC, NORD PL or MAC assembly.

Sometimes you may want to debug a subroutine or function unit in incremental mode. Then the main program must be converted into BRF format and loaded into the system rather than compiled in the usual way. An example will illustrate this.

Suppose your program system consists of the main program (on file MAIN) referencing two external functions F1 (on file F1) and F2 (on file F2). Something is going wrong in F1 and you want to control the execution of it in incremental mode by breaking through it. This system configuration is generated in the following way:

```
COM MAIN, 0, "MAIN"
COM F2, 0, "F2"
OLD F1
LOAD MAIN
```

7.8 *REAL TIME (RT) PROGRAM STATEMENT*

By using the RT program statement you can generate an RT description for your program. This program may be executed in the same way as all other RT programs written in assembly code (see the SINTRAN III User's Guide for further information). The RT statement has the following format:

PROGRAM <prog.name>,<priority>

The <prog.name> may be any acceptable BASIC name. It will be referred to in the loader tables and must be defined only once. The <priority> specifies the priority of the RT program and may be any unsigned integer between 1 and 255. An example might be:

PROGRAM PER, 5%

Here PER will be defined to a real-time program with a priority of 5.

The <priority> may be omitted. Then the <priority> will be set to one, and a warning message will be printed when the program is loaded by the Real Time Loader.

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))
```

7.10.2 *Types of Parameters*

Be sure that actual and formal parameters correspond in type as well as in number. Integers in FORTRAN ought to be integers in BASIC, i.e. postfixed by % or declared in an INTEGER statement. Note that apparent integers as 2 or 100 in BASIC are treated as reals while 1% and 200% are equivalent to FORTRAN integers. It is also the user's responsibility to give parameter arrays the correct dimensions in the DIMENSION respectively DIM statements. (String arrays make no sense as parameters to FORTRAN.)

7.10.3 *Types of Functions*

It is also important that function identifiers (names) agree in type. These identifiers are type-declared like ordinary variables.

Example:

```
 10  EXTERNAL FUN
 20  A = FUN(I)
  .
100  END

 10  FUNCTION FUN(I)
 20  INTEGER FUN
 30  FUN = I
 40  END
```

Here, the function FUN will be treated as type real in the call while the function itself returns an integer value. This will cause the value of A to be unpredictable. Type of identifiers can be examined by the IDENTIFIERS-USED command.

7.11 *MIXED BASIC AND ASSEMBLY ROUTINES*

The NORD BASIC Run-time System has been designed to allow an extensive use of mixed BASIC/ASSEMBLY systems. No special heading format of the assembly routines is necessary, but there exist some restrictions upon the use of the B register.

When calling assembly subroutines/functions from BASIC, the value of the B register on leaving the subprogram must not differ from the entering value. (System value.) Moreover, no locations in the B field (B -200_8 through B $+177_8$) must be changed by the subprogram. The L register holds the return address.

7.11.1 *Parameter Access in Subprograms*

When entering any assembly subprogram, the A register points to a string of the actual parameter addresses (if any).

7.11.2 *Functions in Assembly*

A function must always return with a value, and this must be contained in the central registers.

Integer functions	:	Value in the A register
Double functions	:	Value in the A–D registers
Real functions	:	Value in the T–A–D registers
String functions	:	String descriptor in A–D registers

7.11.3 *Example of a Subprogram Structure*

```
)9BEG
)9ENT SUBR

SUBR,   SWAP SA DB
        STA SAVB        % SAVES B REGISTER
        -
        -
        LDF I 0,B       % ACCESS OF 1. PARAMETER
        -
```

```
        LDF I N-1,B     % ACCESS OF N'TH PARAMETER
        -
        LDA SAVB
        COPY SA DB
        EXIT            % RETURNS TO BASIC

SAVB,0
)9END
```

7.11.4 *Calling a BASIC Subprogram from Assembly*

The calling sequence is explained through the following example:

```
)9BEG
)9EXT 8ENTR SUBR
-
-
JPL I   (8ENTR          % 8ENTR IS A RUN TIME TRANSITION
                        % ROUTINE
        SUBR            % BASIC SUBROUTINE NAME
          N             % NUMBER OF PARAMETERS
        PARAM1          % ADDRESS OF 1. PARAMETER
        -
        -
        PARAMN          % ADDRESS OF N'TH PARAMETER
        -               % RETURN, FUNCTION VALUE IF ANY
        -               % IN RESPECTIVE REGISTER(S)
)FILL
)9END
```

APPENDIX A

SUMMARY OF ERROR MESSAGES

Since it is beyond all expectations that everyone will always write perfect BASIC programs, the language has been provided with a set of informative error messages. These are divided into two main types: messages from the compiler and messages from the run-time system. Generally, the compiler errors arise from incorrect format in a statement or incorrect reference to variables. The messages from the run-time system, which arise during execution, are more varied.

A.1 *COMPILER ERROR MESSAGES*

These messages are self-explanatory as far as it is possible. In the list below they are supplied with some information and examples.

NOT RECOGNIZED

Statement or command not recognized.

```
10 XXX
*** ERROR IN LINE 10 NOT RECOGNIZED
```

ILL. STATEMENT AFTER IF – THEN

The following statements listed are illegal after IF – THEN:

DATA, DIM, DIM#, EXTERNAL, FNEND, FOR, NEXT, FUNCTION, IF, DEFFN, PROGRAM, END, SUBROUTINE

```
10 IF I=0 THEN FNEND
*** ERROR IN LINE 10 ILL. STATEMENT AFTER IF–THEN
```

SYNTAX ERROR

An identifier or delimiter appears in a relationship where it does not make sense to the compiler. The illegal item is referenced in quotation marks.

```
10 A=A++A
*** ERROR IN LINE 10 "+" SYNTAX ERROR

20 FOR 2=1 TO 10
*** ERROR IN LINE 20 "CONSTANT" SYNTAX ERROR
```

MAX TWO SUBSCRIPTS PERMITTED

A matrix declaration reference appears with more than two subscripts.

```
10  B=MAT(I,J,L)
*** ERROR IN LINE 10 "MAT" MAX TWO SUBSCRIPTS
    PERMITTED
```

(MISSING

A matrix declaration must be followed by a dimensioning parenthesis.

```
10  DIM MAT,
*** ERROR IN LINE 10 "," ( MISSING
```

END MISSING

The END statement must be the last statement of a program unit. If not present, the program will not execute.

LINE NUMBER MISSING

May occur during execution in incremental mode if the statement referred to in an ON ERROR GOTO is removed.

NOT STRING TYPE

The identifier involved must be string type.

```
PRINT USING A,B
"A" NOT STRING TYPE
```

MIXED DATATYPES

The left and right side of an operator do not match in type.

```
IF A=A$ THEN 20
"A" MIXED DATATYPES
```

ILL. STRING TERMINATION

A string is not terminated with quotation mark.

```
A$="ABC
ILL. STRING TERMINATION
```

TOO MANY NEW VARIABLES SINCE LAST RUN

This message may occur in immediate mode when a lot of new identifiers are introduced. Try a (dummy) run of the program. (A program may consist of an END statement only.)

ILL. TYPE IN DATA-STATEMENT

Expressions are not permitted in DATA statements.

```
10  DATA 10, 10+20
*** ERROR IN LINE 10 "EXPRESSION" ILL.TYPE IN
    DATA-STATEMENT
```

MAX FOR–LOOP NESTING

These may be nested to maximum depth of 16.

FOR–LOOPS WITHOUT NEXT–MATCH

A FOR statement occurs while the corresponding NEXT statement is missing.

```
10  FOR I=1 TO 50
20  END
RUN
"LINE 20" FOR–LOOPS WITHOUT NEXT–MATCH
```

NEXT–STM. WITHOUT FOR-MATCH

A NEXT statement occurs while the corresponding FOR statement is missing.

```
10  NEXT A
20  END
RUN
"LINE 10" NEXT–STM, WITHOUT FOR–MATCH
```

IMPROPERLY NESTED FOR–LOOPS

FOR–NEXT statements must match in a way outlined in Section 2.3.

```
10  FOR I=1 TO 10
20  FOR J=1 TO 10
30  NEXT I
40  NEXT I
50  END
RUN
"LINE 30" IMPROPERLY NESTED FOR–LOOPS
```

NOT LOGICAL EXPR IN IF

The expression within IF and THEN must have a logical value resulting from arithmetic relations.

```
IF A+1 THEN 20
"A" NOT LOGICAL EXPR IN IF
```

THEN/GOTO/GOSUB NOT FOUND

One of these BASIC language words does not appear in a statement where it is expected to be found. (IF, IF END, ON).

```
ON X PRINT 5
THEN/GOTO/GOSUB NOT FOUND
```

GOTO/GOSUB/THEN – NOT FOLLOWED BY LINE NUMBER

A line number is expected to appear in this relationship.

```
GOTO X+Y
"X" GOTO/GOSUB/THEN – NOT FOLLOWED BY LINE
NUMBER
```

RECURSIVE CALL

Recursive call in basic subprograms is not permitted.

```
10  FUNCTION FUN(I)
20  X=FUN(I)
*** ERROR IN LINE 20 "FUN" RECURSIVE CALL
```

ILL. CHARACTER

A nonstandard character occurs in this line.

```
LET X=!
ILL. CHARACTER
```

LINE <ddd> MISSING

The referenced line is missing and must be added to the program in order to run it.

```
GOTO 50
"LINE 50" MISSING
```

ILL. EXPR IN ON

Something is wrong with the expression in an ON statement.

```
10  ON A$ GOTO 10,20
*** ERROR IN LINE 10 ILL. EXPR IN ON
```

TO MISSING

The word TO must be present in a FOR statement. The word STEP, however, is optional.

```
10  FOR X=1
*** ERROR IN LINE 10 TO MISSING
```

NESTED IF STATEMENTS

An IF statement cannot appear as a conditional statement of another IF.

STRING NOT PERMITTED

The referenced string identifier/constant is not permitted in this relationship.

```
10  DIM ARRAY (A$,100)
*** ERROR IN LINE 10 "A$" STRING NOT PERMITTED
```

OPERATION UPON ILL. DATATYPE

The referenced operand does not make sense in this application.

```
LET A=B+STRING$
"A" OPERATION UPON ILL. DATATYPE
```

INCORRECT NUMBER OF ARGUMENTS

The referenced library function is attempted to be called with incorrect number of arguments.

```
PRINT SIN(A,I%)
"SIN" INCORRECT NUMBER OF ARGUMENTS
```

INCORRECT ARGUMENT TYPE

The referenced function is attempted to be called with incorrect argument type.

```
PRINT SIN(A$)
"A$" INCORRECT ARGUMENT TYPE
```

LOGICAL MIXED WITH NONLOGICAL

A logical expression is involved as one of the operands in an operation while the other one is not.

```
LET ALFA=BETA<DELTA
"ALFA" LOGICAL MIXED WITH NONLOGICAL
```

UNBALANCED PARENTHESIS

The number of left and right parentheses on a line deviate from each other, or they are improperly nested.

```
10  LET INTX=QUOT+(VAR1/(I–SIN(VAR1))
*** ERROR IN LINE 10 UNBALANCED PARENTHESIS
```

STRING ASSIGNMENT ERROR

A string cannot be assigned to a numeric variable and a numeric expression cannot be assigned to a string variable.

```
20  LET STRING$=ALFA
*** ERROR IN LINE 20 "STRING$" STRING ASSIGMENT ERROR
```

MATRIX NOT DIMENSIONED

May occur in immediate mode. The referenced matrix must be declared in a DIM statement.

```
MATPRINT X
"X" MATRIX NOT DIMENSIONED
```

MISSING SUBSCRIPT

The referenced matrix identifier must have subscript(s) in this relationship.

```
10  DIM ARRAY(200)
20  LET X=ARRAY
*** ERROR IN LINE 20 "X" MISSING SUBSCRIPT
```

DEF-FNEND MATCH ERROR

Improper sequencing of DEF and FNEND statements.

```
 10  FNEND
100  END
 RUN
 "LINE 10" DEF-FNEND MATCH ERROR
```

MATRIX/VARIABLE NAME CONFLICT

An identifier cannot be used as single variable somewhere and matrix elsewhere.

```
10  A=20
20  DIM A(20)
*** ERROR IN LINE 20 "A" MATRIX/VARIABLE NAME
    CONFLICT
```

LINE NUMBER USED PREVIOUSLY

May occur while compiling into BRF format. Then two lines with identical numbers are not permitted.

```
COM 1,0,OBJECT
10  REM THIS IS A PROGRAM ....
10  REM....
*** ERROR IN LINE 10 "LINE 10" LINE NUMBER USED
    PREVIOUSLY
```

MISPLACED STATEMENT

PROGRAM, SUBROUTINE, FUNCTION, type declaration and other nonexecutable statements ought to be the first entered to the compiler (lowest line no.). Try the RECOMPILE command or change the line number(s).

```
10  I%=177777B
5  SUBROUTINE OLA
***ERROR IN  LINE 5 MISPLACED STATEMENT
```

TOO MANY OPERANDS

Operand stack overflow. Try to split the statement in two.

OPERATOR STACK OVERFLOW

Try to split the statement in two.

VIRTUAL MATRIX NOT PERMITTED

The referenced matrix identifier is not permitted in this relationship.

```
10  DIM #1: ARRAY1(1000,1000)
20  MAT INPUT ARRAY1
*** ERROR IN LINE 20 "ARRAY1" VIRTUAL MATRIX
    NOT PERMITTED
```

COMMON IDENTIFIER NOT PERMITTED

Due to limitations in usage of COMMON variables.

NEEDS CORRECTION

The referenced statement must be corrected (or deleted) in order to run the program.

```
10  PRINX A,B
*** ERROR IN LINE 10 NOT RECOGNIZED
20  EMD
*** ERROR IN LINE 20 NOT RECOGNIZED
RUN
"LINE 10" NEEDS CORRECTION
"LINE 20" NEEDS CORRECTION
LIST
10  PRINX A,B
20  EMD
```

NOT IN STOP

The CONTINUE command may be applied in the following states only:

- A STOP statement has been reached.
- At a break-point

FATAL COMPILER SYSTEM ERROR <ddd>

A fatal error during compilation has occurred due to hardware or software errors. The octal number, <ddd>, indicates an address in the compiling system.

BRF-CHECKSUM ERROR

The BRF file contents are damaged due to hardware or software errors occurring when it was written or read.

COMMON BLOCK EXPANDED

The length of an already defined common block is declared larger in a subsequently loaded program.

DOUBLY DEFINED

The symbol being defined (either by loading a file or by the DEFINE command) has already been assigned a value.

ILL BRF-CONTROL NO

Non-interpretive information has appeared on the BRF file due to hardware or software errors.

INSUFFICIENT PROGRAM

Error diagnostics have occurred during the compilation process.

ENTRY-TABLE OVERFLOW

The entry-symbol table is filled. Try the TABLE-SIZES command.

NO PROGRAM

The user is trying to start a program with no main module.

AMBIGUOUS

The last command word is abbreviated until an ambiguity has occurred.

? Something is wrong with the command argument(s).

COMPILER TABLE OVERFLOW

The compiler main table has overflowed. Try the TABLE-SIZES command.

MISSING ENTRY

When trying to start a program with undefined entries, the entries are printed preceding this message.

AT UPPER LIMIT

The current load address has reached the absolute upper limit or the beginning of the common area.

LINE NUMBER OVERFLOW

1 > line number or line number > 32767 in RENUMBER. The program is **intact** and no renumbering took place.

MAXIMUM LINE LENGTH EXCEEDED

1. RENUMBER expands line beyond limit (125 characters). The renumbering of the program was executed, but the referenced line was truncated and marked erroneous.

   or

2. Too long source line on input. The line w as skipped.

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 message may be printed as 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

APPENDIX B

SUMMARY OF ELEMENTS

This chapter contains lists and short descriptions of the main elements with which the NORD–10 BASIC language is built.

B.1 *STATEMENTS*

The part of the statement in capital letters must appear as it is written, i.e. no abbreviation is allowed. The part of the statement enclosed in < > is supplied by the programmer.

CALL <name>(<parameter list>)

CALL transfers control to an external subprogram written in BASIC/FORTRAN/NPL/MAC. This is a call by reference, i.e. the actual parameter addresses are passed to the subprogram.

CLOSE # <expression>:

The expression denotes a connect device number which is associated with a file in an OPEN statement. The file is closed.

CHAIN <string expression>

Automatic loading and starting of the BRF program unit(s) indicated by the file name(s).

COMMON [/[<Block>]/]<variable>[([<subscript string>])] [=<length>] , ...

A program may be divided into independently compiled subprograms that use the same data. The common statement reserves storage areas (blank or labelled) that can be referenced by more than one subprogram written in BASIC, FORTRAN, NPL or MAC assembly.

DATA <data list>

DATA statements supply one or more numbers or strings to be assigned to variables through a READ statement in a program. Individual items in the data list are separated by commas; strings containing special characters such as commas, ampersands, leading blanks, apostrophes, and strings which begin with a digit, +, or – must be enclosed in quotation marks.

DEF <function definition>

By using this statement, the user may define his own functions. When DEF is followed simply by the function name and the dummy arguments enclosed in parentheses, this statement marks the beginning of a multiple line function definition.

DIM <variable name (dimension(s)), ...>

This statement reserves space for arrays used by a program. BASIC implicitly reserves room for entries zero through 10 (11 spaces total), or entries in columns and rows zero through 10 (121 spaces total).

DIM # <expression>:<list>

The expression denotes the connect device number which is associated with a random access file in an OPEN statement. The list must appear as it would in a standard DIM statement except that a maximum string length may be supplied.

DOUBLE <list of variables>

The variables separated by commas are declared to be of double integer type. This is a non-executable statement which should be specified prior to any executable statement.

END

This statement marks the end of a BASIC program. It must be present and must be the statement with the largest line number in the program. END specifies the return point of subprograms.

EOF

Marks the termination of compiling several program units to BRF.

EXTERNAL <list of functions>

External functions must be declared in an EXTERNAL statement before the function references. Non-executable statement.

FNEND

The FNEND statement marks the end of a multiple line function definition.

FOR <running variable> = <A> TO <B>[STEP <C>]

where A is the FROM element
B is the TO element
C is the STEP element

This statement marks the beginning of a FOR–NEXT loop and is always used in conjunction with a NEXT statement. The running variable must be a nonsubscripted numeric variable. The FROM, TO, and STEP elements may be any arithmetic expression and they need not be integers. The statements between the FOR and NEXT statements will be executed repeatedly subject to these conditions: the first time through the loop, the running variable will have the value specified by the FROM element, and its value will change with each pass through the loop in increments specified by the STEP element; this looping process will continue until the value of the running variable exceeds the TO element. The STEP element is assumed to be one unless otherwise specified.

FUNCTION <name>(<parameter list>)

Declares a FUNCTION subprogram unit. The type of the function is determined by its name which again determines the value of the function being returned in the execution of the END statement. The parameter list contains the formal parameters.

GOSUB <line number>

This statement causes the computer to execute a block of statements in the program called a subroutine; the line number given should be the line number of the first statement in the subroutine. RETURN ends the subroutine.

GO TO <line number>

A GO TO statement causes the computer to take as its new instruction the statement specified by the line number given.

IF <logical expression> GOTO / GOSUB / THEN <line number> or

IF <logical expression> THEN <statement>:.....<statement>

Dependent on the logical expression being true or false the program will take the action following THEN/GOTO/GOSUB or skip to the next line.

IF END # <expression> THEN <line number>

The line number is stored in the file table entry of the file associated with the expression. Later, if end of that file is encountered, BASIC will start the user written subroutine at the specified line number.

INPUT [#<connect device identifier>:] <list>

Requests the input of an amount of data from the terminal, a sequential file or a string. If there are more data on a line than the INPUT statement calls for, the excess data are ignored. If there are insufficient data on the line, the program looks for more data on the next line. A ? is printed for each line entered from the terminal.

INTEGER <list of variables>

The variables separated by commas are declared to be of integer type. Non-executable statement.

LET <A> = <B> [... = <Z>]

A, B, ... are numeric or string variables and Z is an arithmetic expression or string expression. The LET instruction assigns the value of the expression to the variables specified. All items must be either string or numeric.

LINPUT [<connect device identifier>:] <list>

This statement requests the input of a number of strings equal to the number of string variables in the list. Special characters such as commas, quotation marks, leading blanks etc., may be part of the strings entered, A new question mark appears on the terminal for each name in the list. Input may also come from a sequential file or a string.

MARGIN [<connect device identifier>:] <expression>

Sets the maximum number of characters on one line which may be printed on the terminal or a sequential file.

MAT

This sequence of characters designates the statement as one designed to perform operations on whole arrays. The MAT statements are described in a separate portion later in this section.

NEXT <numeric variable name>

The NEXT statement is always used in conjunction with a FOR statement and marks the end of the loop designated by this pair of statements. The variable name must be exactly the name designated for the running variable in the corresponding FOR statement.

ON <expression> GOTO / GOSUB <list of line numbers>

That is what is sometimes called a switch and transfers control to the first line number if the value of the arithmetic expression is one, to the second line number if the value is two, and so on.

GOSUB transfers control to the subroutine beginning with the first line number if the value of the arithmetic expression is one, to the second line number if the value of the expression is two, and so on, When a RETURN statement is encountered in the subroutine, control passes to the statement following the ON–GOSUB statement.

ON ERROR GOTO <line number>

No GOTO action is taken when executing this statement. However, if an error occurs later on, the user written error routine is started at the line number indicated without printing any message.

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

The expression denotes the connect device which is associated with the file name. Access mode describes how the file is going to be used later. OPEN is valid until the corresponding CLOSE statement or until execution is interrupted by the ESC character.

PRINT [#<connect device identifier>:] <list>

The PRINT instruction causes all constants and the current values of all variables or expressions in the PRINT list to be printed on the terminal, sequential file or string.

Appropriate delimiters separate the individual items in the PRINT list. A comma causes the terminal type head to space to the next fifth of the line before continuing the printout; a semicolon causes items to be printed in closely packed format.

[PRINT] USING[#<connect device identifier>:] <string expression>,<list>

The items in the list are printed according to the format specified in the USING string. PRINT USING may output to the terminal a sequential file or a string.

PROGRAM <name>[,<priority>]

May be used as the first statement of the main program, but is optional. The name may be any alphanumeric identifier from one to six characters. Priority is valid for real-time programs only.

RANDOM

Causes the function RND to supply a random number when it is called.

READ <list>

A READ statement causes the next available number in a DATA statement to be assigned to the first numeric variable in the READ list and the next available string in a DATA statement to be assigned to the first string variable in the READ list and so on until all variables in the READ list have been assigned values.

REAL <list of variables>

The variables separated by commas are declared to be of real type. Nonexecutable statement.

REM <any characters>

If the first three characters following the line number of a BASIC statement are REM then any remarks may follow on that line. BASIC ignores this statement.

REPEAT <expression> [STEP<expression>] :<statement>...<statement>

The value of the @ variable is set to one which is later incremented by one if STEP is omitted. The following statements on the same line will be repeated until @ exceeds the first expression which specifies the maximum.

RESET [* or $]

Resets the pointers for string and numeric data in a DATA statement. RESET* resets the pointer for numeric data while RESET$ resets the pointer for string data only.

RETURN

When encountered in a subroutine, this statement returns control to the statement following the calling GOSUB statement. It is emphasized that this statement must not be confused with the identical FORTRAN statement.

STOP

Halts the program execution.

SUBROUTINE <name> [(<parameter list>)]

Must be the first statement of a SUBROUTINE subprogram. Name is an alphanumeric identifier. The parameter list is optional.

WRITE [#<connect device identifier>:] <list>

The expressions in the list separated by commas are output to the terminal, sequential file or string. The elements are separated by commas. — This line is readable by a matching INPUT statement.

MAT STATEMENTS

In the following a description of each MAT statement available in NORD–10 BASIC is given. These statements operate on entire ment which is enclosed in the symbols < > is included when that MAT statement is used with a file. N represents a connect device identifier denoting a file or even a string when simulating sequential files.

MAT C = A+B

This statement adds arrays A and B and stores the result in C.

MAT C = A–B

This statement subtracts array B from array A ans stores the result in C.

MAT C = A*B

This statement causes array C to be set up as the product of A and B.

MAT A = (K)*B

Each entry in B is multiplied by the value of K to form the corresponding entry of array A. K is any arithmetic expression and must be enclosed in parentheses.

MAT A = CON

This statement sets all entries in array A to one.

MAT A = IDN

This statement sets A equal to the identity matrix.

MAT A = INV (B)

Matrix A becomes the inverse of matrix B.

MAT A = TRN (B)

Matrix A becomes the transpose of B.

MAT A = ZER

All entries in array A are set to zero;

MAT A = B

Each entry in A is set equal to the corresponding entry in array B.

MAT INPUT <#N:> A, B, C

A variable amount of data may be input for each singly subscripted array in the list. If input is from the terminal, a new question mark appears for each array in the list. If a file is used, it must be sequential, and the data must be in the same format used if they were input from the terminal. The arrays are filled by rows.

MAT LINPUT <#N:> A$, B$, C$

This statement inputs strings possibly containing commas, ampersands, or other special characters. It reads one line for each entry in the arrays listed. A variable amount of input is not allowed. A new question mark appears on the terminal for each entry of each array.

MAT PRINT <#N:> A, B; C

This statement prints the arrays A and C in regular matrix format and B in closely packed format

MAT [PRINT] USING <#N:> X$, A, B

The arrays A and B are printed according to the format specified in the string X$. Each new row of a doubly subscripted array is put at the beginning of a new line.

MAT READ A, B, C

This statement reads data from DATA statements in the program into the arrays A, B and C. The arrays may be string arrays and information is stored by rows.

MAT WRITE <#N:> A, B

Array layout as with MAT PRINT, except that the elements always are closely packed and separated with commas. Thus, arrays output to sequential files should be input by matching MAT INPUT statements.

B.2 *COMMANDS*

Commands as opposed to statements may be abbreviated until ambiguity occurs. The complete command words are listed here. Command parameters may generally not be abbreviated unless the parameter is a file name. The HELP command gives a list of all the available commands.

BREAK <line number>

When the specified statement is reached, the execution will halt and the control will be transferred to the BASIC processor. The break is performed before the specified statement is executed.

In this state a new break may be specified and the execution continues by using the CONTINUE command. The use of breaks combined with immediate mode provides a powerful and simple debugging aid.

BYE

The control is transferred from the BASIC subsystem to the operating system and any files opened are closed.

CLC <number>

Special for system debugging purposes only.

CLEAR

Clears the user's area in memory and resets the system to its initial state. However, system parameters which are especially affected by commands will not be reset (IGNORE-MATRIX-CHECK, TABLE-SIZES, etc.).

COMPILE <source file>[<list file><BRF object file>]

Starts compilation of the program which resides on the source file. If the second parameter is present a listing of the program is obtained on the list file/device specified, error messages as well.

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>] or <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 ND-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.

SAVE <file name>

The current program will be saved (symbolic version) on the file specified.

SET-LOAD-ADDRESS <octal address>

Subsequent loading will start from the address specified.

TABLE-SIZES <octal number> <octal number>

If one of the error messages ENTRY-TABLE OVERFLOW or COMPILER TABLE OVERFLOW occurs, it is possible to change the size(s) by this command. The first argument gives the number of *entries* in the ENTRY table. The second gives the number of *locations* in the COMPILER table. Initial values are 200 and 10000. This command implies a CLEAR.

UPPER-LIMIT <octal address>

The user load area upper limit is set to the value specified.

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

Exclusive load. Library sequences headed with *defined* symbols are skipped while all other units on the file specified will be loaded until EOF (control byte 23) is encountered. Default file type is: BRF. This command is somewhat special and is used for system generation.

B.3 *FUNCTIONS*

Some functions are already described in the text of this manual. Here is given a complete list with short descriptions of all available BASIC functions. The functions are divided into three groups: Mathematical-, String- and Miscellaneous functions. Functions belonging to the Extended Library (not built-in, but available on a file) are given a special mark.

The variables used in the examples indicate the *type* of function and parameter(s) which are optimal regarding execution time and memory space. However, integer, double integer and real may replace each other.

Mathematical Functions

Extended Library	Function	Explanation
	Y=ABS(X)	returns the absolute value of X
	Y=ATN(X)	returns the arctangent of X in radians
	Y=COS(X)	returns the cosine of X in radians
	Y=EXP(X)	returns the value of e↑X, where e=2.71828182
	Y=SIN(X)	returns the sine of X in radians
	Y=SQR(X)	returns the square root of X
	Y=TAN(X)	returns the tangent of X in radians
	Y=LOG(X)	returns the natural logarithm of X, $\log_e X$.
	Y=LOG10(X)	returns the common logarithm of X, $\log_{10} X$
	Y=PI	has a constant value of 3.14159265
	Y=SGN(X)	returns +1 if X>0, 0 if X=0 and −1 if X<0
	Y=INT(X)	returns the largest integer not greater than X
x	Y=ASI(X)	returns the arcsine of X in radians
x	Y=ACO(X)	returns the arccosine of X in radians
x	Y=DEG(X)	returns the value of X radians in degrees
x	Y=RDN(X)	returns the value of X degrees in radians
x	X=CAX(R,A)	returns cartesian X-coordinate of the polar representation, R(radius), A (angle in radians).
x	Y=CAY(R,A)	returns cartesian Y-coordinate of the polar do.do.

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

Tab

I^C Causes space to the next tab stop. Rings the bell and takes no further action if there are no more tab stops on the line. The tab stops are: 8, 14, 30, 40, 50, 60, 70, 80.

Line Terminate

M^C Control M, which is really carriage return, serves to terminate an edit and delimit lines of text. Automatically supplies a line feed when M^C is typed.

Escape Character

$V^C\underline{C}$ Causes the character $\underline{C}$ to be appended literally to the text being collected and disables any other function $\underline{C}$ might have. $\underline{C}$ may be any character.

Backspace Characters

The following control characters delete one or more characters from the end of a line being typed in. All but Q^C may be iterated. Rings the bell if empty.

A^C Prints ↑ and deletes the preceding character. A^C does not affect the status of the old line.

W^C Prints \ and deletes the preceding "word". I.e., all preceding blanks are deleted, and all non-blank characters up to the next preceding blank.

Q^C Prints ← followed by a carriage return line feed, and deletes the line currently being typed in.

Copy Characters

The following characters copy one or more characters from the old line onto the end of the new line. Rings the bell (except in D^C and F^C) if the old line contains no more characters or if the character to be copied to does not exist in the old line.

C^C Copies the next character of the old line onto the new line and prints out the character copied.

U^C Copies characters from the old line onto the new line up to the next tab stop and prints out the characters copied.

H^C Copies the rest of the old line onto the new line, printing out characters copied. Editing may now continue.

$O^C\underline{C}$ Copies the old line up to, but not including, the next occurrence of the character $\underline{C}$, printing out characters copied.

$Z^C\underline{C}$ Copies the old line up *through* the next occurrence of the character $\underline{C}$ in the old line, printing out characters copied.

D^C Copies the rest of the old line onto the new, printing out the characters copied. The edit is also terminated. D^C is equivalent to H^CM^C.

F^C Copies the rest of the old line onto the new line *without* printing it. The edit is also terminated.

Skip Characters

The following control characters cause one or more characters from the old line to be skipped. The new line is not affected. Prints % for each character skipped. Rings the bell if there are no more characters in the old line or if (in P^C and X^C) the character to be skipped to does not exist in the rest of the old line.

S^C Skips the next character of the old line.

$P^C\underline{C}$ Skips up to, but not including the next occurrence of the character $\underline{C}$ in the old line. (P^C is the analogue of O^C).

$X^C\underline{C}$ Skips up *through* the next occurrence of the character $\underline{C}$ in the old line. (X^C is the analogue of Z^C.)

Reprint Characters

The following control characters *do not* affect the state of the edit (i.e., the contents of the old line and new line). They merely permit the user to recover in case he has become confused about the state of the edit. Editing may thereafter continue normally.

R^C Prints line feed and then the rest of the old line. On the next line is printed out the new line produced so far.

T^C Prints out the state of the edit as in R^C, except that the rest of the old line is properly aligned with the new line.

The Re-Edit Character

Y^C Copies, without printing, the rest of the old line onto the new line and then gives a carriage return line feed. Now the resulting line may be edited.

Mode Characters

E^C Changes the mode from replace to insert, and prints <, or from insert to replace, and prints >. The mode is replace at the beginning of each line. In replace mode characters typed by the user replace those of the old one-for-one. In insert mode characters typed by the user are appended to the new line without affecting the old line. Skips and copies work as before.

L^C Acts exactly like carriage return or M^C. In other words, L^C serves to terminate text input.

C.4 *ASCII CHARACTER SET*

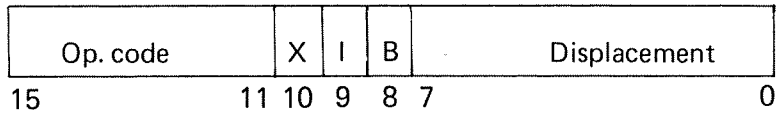
Graphic	Octal Value	Decimal Value	ASC Abbreviation	Comments
	0	0	NUL	Null
	1	1	SOH	Start of heading
	2	2	STX	Start of text
	3	3	ETX	End of text
	4	4	EOT	End of transmission
	5	5	ENQ	Enquiry
	6	6	ACK	Acknowledge
	7	7	BEL	Bell
	10	8	BS	Backspace
	11	9	HT	Horizontal tabulation
	12	10	LF	Line feed
	13	11	VT	Vertical tabulation
	14	12	FF	Form feed
	15	13	CR	Carriage return
	16	14	SO	Shift out
	17	15	SI	Shift in
	20	16	DLE	Data link escape
	21	17	DC1	Device control 1
	22	18	DC2	Device control 2
	23	19	DC3	Device control 3
	24	20	DC4	Device control 4
	25	21	NAK	Negative acknowledge
	26	22	SYN	Synchronous idle
	27	23	ETB	End of transmission block
	30	24	CAN	Cancel
	31	25	EM	End of medium
	32	26	SUB	Substitute
	33	27	ESC	Escape
	34	28	FS	File separator
	35	29	GS	Group separator
	36	30	RS	Record separator
	37	31	US	Unit separator
	40	32	SP	Space
!	41	33	!	Exclamation point
"	42	34	"	Quotation marks
#	43	35	#	Number sign
$	44	36	$	Dollar sign
%	45	37	%	Percent sign
&	46	38	&	Ampersand
'	47	39	'	Apostrophe
(	50	40	(	Opening parenthesis

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

C.5 *NORD WORD STRUCTURE*

Instruction Word



One instruction word always occupies one location, 16 bits, of main memory. The operation code occupies the five most significant bits (11 – 15), and specifies one of 32 instructions.

For memory reference instructions bits 0 – 10 are used to specify the address of the instruction. The instructions which do not have an address use the bits for further specification. Bits 8, 9 and 10, called ,B I and ,X are used to control the address computation.

The displacement is an 8 bit signed number ranging from –128 to +127, using two's complement for negative numbers and sign extension to produce a 16 bit number.

Data Word

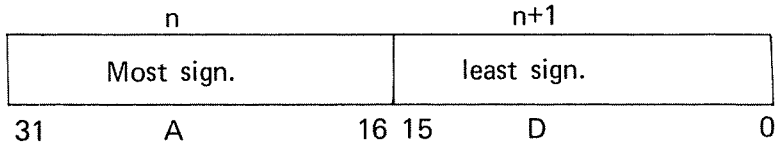
Three basic types of data words exist:

a) Single length numbers:

A 16 bit number which occupies one memory location. Representation of negative numbers are in two's complement. Range as integers: $-32768 \leqslant x \leqslant 32767$.

b) Double length numbers:

A 32 bit number which occupies two consecutive locations in memory, and where negative numbers also are in two's complement.



A double word is always referred to by the address of its most significant part. Normally a double word is transferred to the registers so that the most significant part is contained in the A register and the least significant in the D register. Range as integers: $-2\ 147\ 483\ 648 \leqslant x \leqslant 2\ 147\ 483\ 647$.

c) Floating point numbers (48 bits):

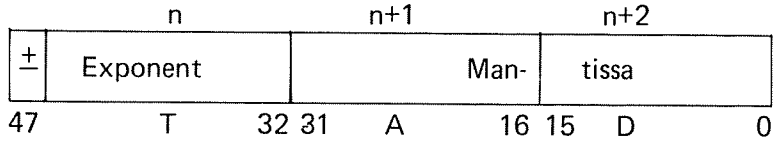
The data format of floating point words is 32 bits mantissa magnitude, one bit for the sign of the number and 15 bits for a signed exponent.

The mantissa is always normalized, $0.5 \leqslant$ mantissa < 1; for all non-zero numbers bit 31 equals one. The exponent base is 2. The exponent is biased with 2^{14}, i.e., 40000_8 is added to the actual exponent, so that a standardized floating zero contains zero in all 48 bits.

In memory one floating point data word occupies three 16 bit locations, which are addressed by the address of the exponent part.

n	exponent and sign
n+1	most significant part of mantissa
n+2	least significant part of mantissa

In CPU registers bits 0 – 15 of the mantissa are in the D register, bits 16 – 31 in the A register, and bits 32 – 47, exponent and sign, in the T register. These three registers together are defined as the floating accumulator.



The accuracy is 32 bits or approximately 10 decimal digits, any integer up to 2^{32} has an exact floating point representation. The range is:

$$2^{-16384} * 0.5 \leqslant |x| < 2^{16383} * 1 \text{ or } x = 0$$

or

$$10^{-4931} < |x| < 10^{4931}$$

Examples (octal format):

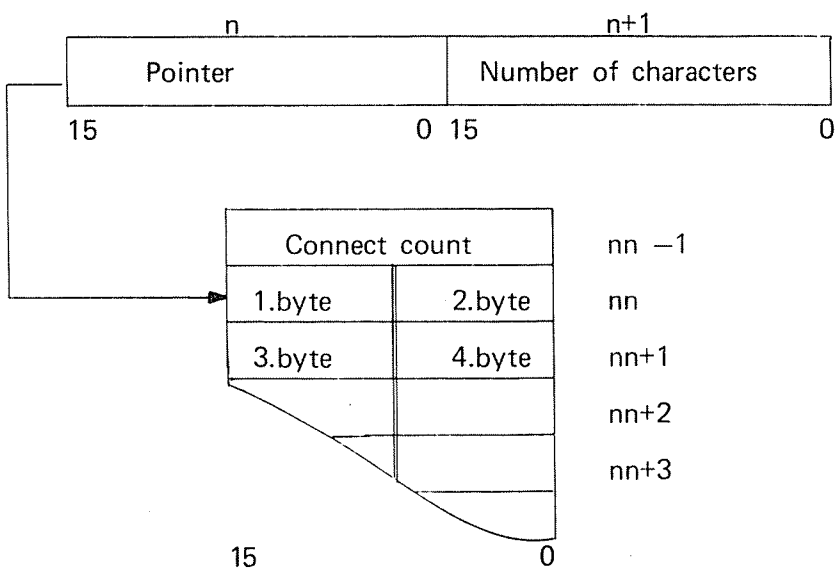
	T	A	D
0:	0	0	0
+1:	040001	100000	0
–1:	140001	100000	0

Any other data word format than those four described here may be programmed. These four data word formats have corresponding instructions which make these formats easy and natural to use. It is also rather easy to program data word formats using one bit data word and 8 bit data word because the NORD has instructions giving direct access to single bits and 8 bit bytes.

BASIC has two additional data word formats:

d) Character strings:

The data format of strings consists of a two word object which contains a pointer to the direct memory location of the string and the number of characters in the string. The string itself consists of the ASCII values packed two by two into one word. The words are stored in consecutive order. The parity bit (bit 7) is always set to zero. The string is preceded by one word indicating the number of connections to the string.



e) As the NORD-10 may be supplied with a microprogram which operates on 32 bit real numbers, a special version is available for users who stick to this format.

The library and run-time systems have separate versions for the 48 and 32 bit real arithmetic.

The use of the two versions are identical but the user should consider the small precision (6 - 7 digits) of 32 bit reals. Especially in mixed mode arithmetic because double integers have greater precision!

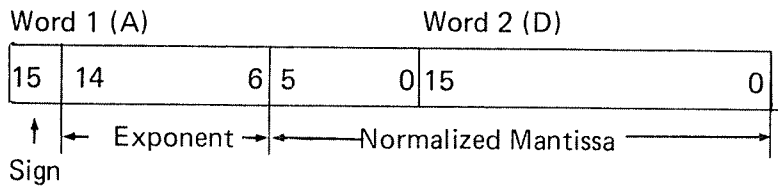
The data format of floating point words is 22 + 1 bits mantissa magnitude, one bit for the sign of the number and 9 bits for a signed exponent.

The mantissa is always normalized, $0.5 \leqslant \text{mantissa} < 1$. The exponent base is 2. The exponent is biased with 2^8, i.e., 400_8 is added to the actual exponent, so that a standarized floating zero contains zero in all 32 bits.

In the computer memory one floating point data word occupies two 16 bit locations, which are addressed by the address of the exponent part.

n	exponent, sign bit and most significant part of mantissa
n + 1	least significant part of mantissa

In CPU registers, bit 0 - 15 of the mantissa are in the D register, bit 16 - 31, the most significant part of the mantissa, exponent and sign, in the A register. These two registers together are defined as the floating accumulator.



The accuracy is 23 bits or 6-7 decimal digits, any integer up to $2^{23} - 1$ has an exact floating point representation.

Note: The one extra bit in the mantissa is the most significant, and is set to one if not all bits in the exponent is zero. It is removed in the result.

The range of a floating point number is approximately -10^{76} through $+10^{76}$

APPENDIX D

INDEX

A/S NORSK DATA-ELEKTRONIKK
Lørenveien 57, Oslo 5 - Tlf. 21 73 71

COMMENT AND EVALUATION SHEET

NORD-10 BASIC – Compiler Reference Manual ND-60.071.01
August 1976

In order for this manual to develop to the point where it best suits your needs, we must have your comments, corrections, suggestions for additions, etc. Please write down your comments on this pre-addressed form and post it. Please be specific wherever possible.

FROM: _____

_____

_____

– we make bits for the future

NORSK DATA A.S BOX 4 LINDEBERG GÅRD OSLO 10 NORWAY PHONE: 30 90 30 TELEX: 18661