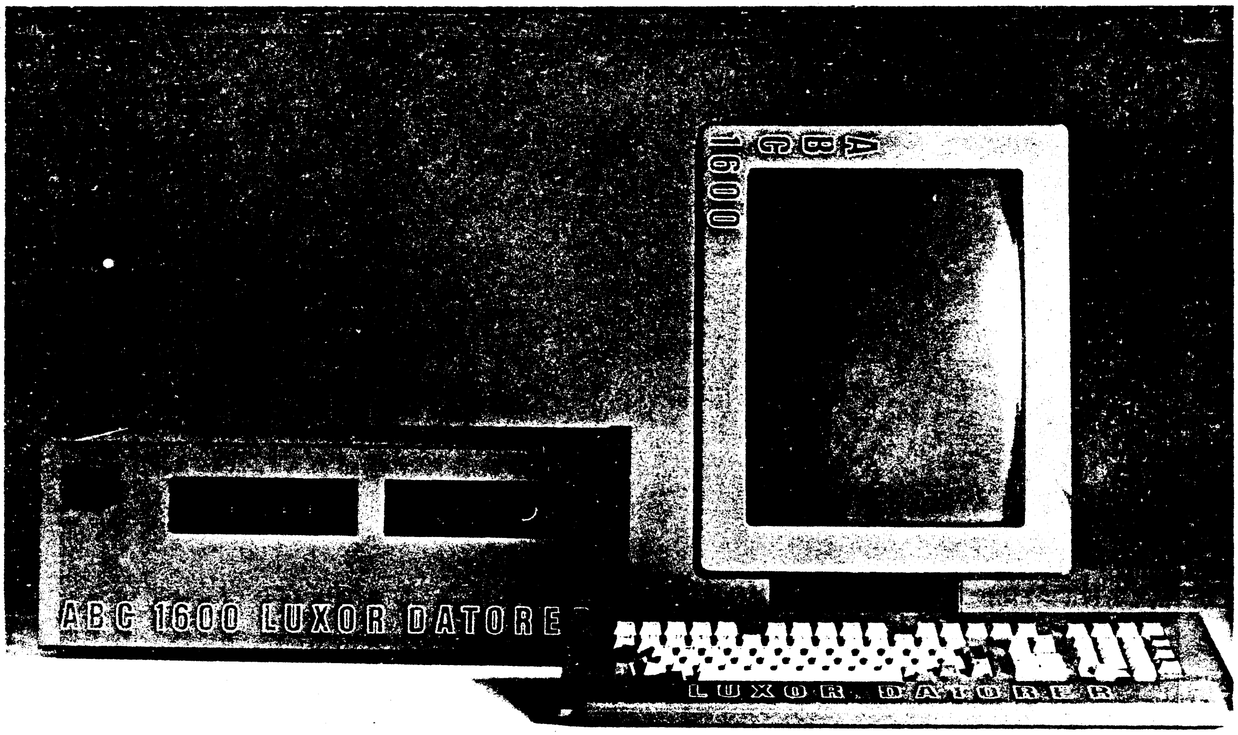
LUXOR

Preliminary Version

ABC 1600
Basic III with MIMER



ABC 1600®

LUXOR
Computers

PURPOSE OF THIS DOCUMENT

This document is a Programmer's Reference Manual. It is to be used by experienced programmers as a reference tool. It is not intended for use as a learning aid by non-programmers.

CONTENTS

LIST OF TABLES

1. INTRODUCTION

Contents

1. INTRODUCTION

1.1 Overview of DIAB BASIC III

This manual describes DIAB's Extended BASIC III hereafter referred to as BASIC III. BASIC III is a comprehensive, commercially oriented, semi-compiled language which is available for computers running D-NIX operating system. It is an implementation of the BASIC language initially developed by Dartmouth College and standardized by the American Standards Institute (American National Standard for Minimal BASIC, ANSI X.360-1978).

It must be emphasized that this is a Programmer's Reference Manual and not a tutorial. Hence, it is designed to be used primarily as a reference device by experienced programmers.

BASIC is one of the simplest of all programming languages because of its small number of powerful but easily understood statements, functions and commands and its easy application to problem solving. Nevertheless, the language is comprehensive enough to allow versatile and efficient solutions to most problems. The wide use of BASIC in scientific, business, and education installations attests to its value and straight forward application.

BASIC III is an interpreter program stored on disk and called into memory by the user when required. An interpreter is a type of compiler which checks or interprets your source program as you enter it line by line. The source program resides in memory along with the interpreter for as long as the user requires and can be saved and run whenever needed. This is in contrast to other compilers which save the computer readable form (i.e. the object program) and then execute the object program whenever needed. Because BASIC III is an interpretive language, a syntax error will result in an immediate error message on the screen. You can also run the program at any time to test portions that have been entered. This is called interactive programming and is in many cases the most efficient way of programming. However, interactive programming does not solve all the problems. When formal errors have been eliminated from the program, logical errors may still remain. These can only be detected when the program is executed with the proper data.

BASIC III contains elementary statements to write simple programs. Advanced programming features and statements are also provided to produce more complex and efficient programs. The keyword here is efficient. Almost any problem can be solved with the simple BASIC statements. Later in the user's programming experience, the advanced techniques can be added.

BASIC III also allows the use of multi-character variable names and free use of comments and spaces, which aid in creating programs that are self-documentering and maintainable

The available BASIC versions and options are described in appendix F.

1.2 Text symbols and conventions

Throughout this manual specific documentation conventions are used to describe formats for writing BASIC III commands, statements, and functions. The following conventions are in effect:

Symbol	Description and Use
1. CAPITAL LETTERS	Capital letters are used for all keywords, commands, functions, and statements that are to be explicitly typed. Example: LIST
2. Lower case	Lower case letters specify variables which are to be supplied by the user according to the rules explained below and in this text. Example: DATA <list>
3. < >	Angle brackets enclose fields that are required for valid Basic III syntax. They are never to be typed unless "not equal to" is to be specified by "<>". Example: LET <variable> = <express> LET A = 4
4. ( ) , -	Parentheses enclose required elements or keywords of a statement. Commas and dashes are separators. All must be typed as shown. Example: COMPZ(A$,B$) LIST 100-400
5. [ ]	Square brackets enclose optional elements of a statement or indicate an optional choice of one element among optional elements. Example: GET <stringvar> [COUNT bytes]
6. ... , ...	Ellipsis (three dots) indicate multiple arguments are allowed. Example: POKE <address>,<data>
7. ®	The symbol "®" indicates the depression of the RETURN key. Example: LIST®
8. ~	The symbol "~" is a BASIC symbol with the ASCII value 126 dec. used in formatted printing in section 5. Example: PRINT USING "##~##" 21.15
9. CTRL-H	Control character. Depress and hold CTRL key while striking another key (represented by H). Example: CTRL C CTRL H

1.3 Character set

BASIC III is designed to utilize the American Standard Code for Information Interchange (ASCII) for its character set. This set includes:

1. Printable Characters
   a. Letters A through Z
   b. Lower case letters a through z
   c. Numbers 0 through 9
   d. Special punctuation characters and symbols

2. Control Characters

Appendix A shows the complete set of BASIC ASCII characters and their respective decimal codes.

1.4 Organization of this manual

This manual is organized into 11 sections and appendices. Sections 2 and 3 contain information necessary to understand and work with BASIC III. There are many types of BASIC in use today. Hence, these sections define the procedures and language elements within the context of DIAB BASIC III language.

Sections 4 through 6 describe the individual control commands, statements, and functions available in BASIC III. Statements are divided into three sections: Data, Input/Output and Program Control statements for ease of reference. For each command, statement, or function, the following information is included:

1.	Function	- Summarizes purpose of statement.
2.	Mode	- Specifies which mode applies - Direct, Program or both.
3.	Format	- Shows the command syntax.
4.	Arguments	- Defines the format variables.
5.	Use	- Describes in detail how the command is used including restrictions and exceptions.
6.	Example	- Lists program examples illustrating the various uses of the command.

Section 7 contains advanced file information and BASIC III statements and functions to be used for sophisticated programming. Detailed knowledge of BASIC III and the operating system is required before this information can be applied.

Section 8 contains information on options to the BASIC III.

Section 11 contains a quick reference summary of BASIC III commands, statements, and functions.

Appendix A shows the complete set of BASIC ASCII characters and their respective decimal codes.

Appendix B contains a list of error messages with comments.

Appendix D describes the differencies between BASIC III and the earlier BASIC versions.

Appendix F describes the available BASIC versions

1.5 Abbreviations

The following abbreviations are used in this manual:

line no.	-	Line Number
fd	-	File descriptor
vol	-	Volume
PR	-	Printer
string'var	-	String Variable
channel no.	-	Channel Number
record no.	-	Record Number

2. WORKING WITH BASIC III

Contents

2. WORKING WITH DIAB BASIC III

2.1 Initiating and terminating Basic III

To load the Basic III interpreter the user entes the word "basic" in response to the operating system promt:

```
$ basic  [-d -x -i -m] [file] ®
(note:    ®=Return key)
```

where the switches have the following meaning:

-d   Set floating precision to DOUBLE

-x   Set EXTEND mode

-i   Set INTEGER mode

-mXX Get XX number of kbytes. Extra memory (default 32 k)

If "file" is specified the file "file" is loaded and executed immeditely by the interpreter.

Example:

Start basic in INTEGER and EXTEND mode:

```
basic -i -x
```

Basic III begins loading and when ready displays:

```
Basic III  Ver X.XX
Copyright Dataindustrier DIAB AB, Sweden 1984

*basic*
```

(X.XX refers to version number and update number.)

During startup, one of the warning messages can appear:

1: NO TERM environment variable.
Meaning: Before startup of basic the shell variable TERM should be set to the terminal type you are using.

Example:
If you are using a vt100 terminal:

```
$ TERM = vt100
$ export  TERM
```

2: No termcap entry for : XX
Meaning: The TERM variable is wrong or there is no entry in the terminal capability database file ("/etc/termcap") for your terminal type.XX. For information about how to create an entry in termcap, refer to your D-NIX manual (termcap (5)).

3: No bascap entry for : XX
Meaning: There is no entry in the file ("/usr/etc/bascap") for your terminal type XX.

In the bascap file you should define the input control characters for the editor. The same syntax as for termcap is used:

Example: For an ADM3a terminal the entry could look like this:

```
laöadm3aö3aölsi adm3a:Ö
     :ku=^K:kd=^N:kr=^L:kl=^H:Ö
     :ei=^U:ic=^I:dc=^E:cd=^D:Ö
     :dl=^X:
```

giving the same keys as default. (See below.)

Note:
If warning 1 or 2 appears, the interpreter works but no editing is possible since it does not know how to move the cursor on your terminal.

If warning 3 apppears, default values are used.

The file "termcap" contains the information about what control characters (or sequences) to SEND to the terminal while the file "bascap" contains the information about what to RECEIVE from the user.

Abrev.	Termcap	Bascap	Default	Meaning
"kr"	x	x	^L	right arrow key
"kl"	x	x	^H	left arrow key
"ku"	x	x	^K	up arrow key
"kd"	x	x	^N	down arrow key
"cm"	x			cursor positioning
"cl"	x			clear display
"ic"		x	^I	enter insert mode
"ei"		x	^U	exit insert mode
"dc"		x	^E	delete character
"cd"		x	^D	delete to EOL
"dl"		x	^X	delete line
"nd"	x			non destructive space
"up"	x			move cursor up
"am"	x		yes	automatic margin
"bw"	x		no	back wrap
"co"	x		80	# of columns

The arrow key entries in bascap overrules the ones in termcap if specified in both.

Note:
In the description of editing functions (sections 2.3, 4.2) the default control characters are used and should be replaced by the character or function key defined in "/usr/etc/bascap" or "bascap".

From the basic, shell commands can be executed by starting the line with an exclamation mark (!). The rest of the line is then passed to a new shell which executes it.

Example 2:
Start basic with 10 kbytes of extra memory (for a total of 42 kbytes)

```
basic  -m10 ®

Basic III  Ver  X.XX
Copyright  Dataindustrier DIAB AB, Sweden 1984

*basic*
;SYS(2)  (memory available)
 42806
*basic*
```

Example 3:
If a basic program

```
10; "Hej"
20  BYE
```

is saved in file "hejpgm.bac". Execute it:

```
$basic .hejpgm
Hej
$
```

To remove Basic III from working memory and return to the operating system type:

```
BYE  ®
```

2.2 Modes of operation

Basic III allows for three modes of operation:

1. Program Mode
2. Direct Mode
3. Run Mode

2.2.1 Program Mode

Basic III distinguishes between statements intended for immediate execution and statements intended for delayed execution. This difference is based solely on the absence or presence of a line number in front of a statement. A statement preceded by a number, for example:

```
180 INPUT X,Y
```

is noted as being intended for delayed execution. Statements discussed in this text to which this definition is applicable have "Program" specified after the Mode declaration.

2.2.2 Direct Mode

Conversely, the absence of a line number in statements such as:

```
        PRINT 1000/12®
```

cause the interpreter program to execute it directly after depression of the RETURN key. This is called the direct mode and so specified for applicable statements in this text.

The Direct mode also allows for immediate solution of problems, generally mathematical, which do not require interactive program procedures. For example:

```
        A=1.5: B=3: PRINT A, B, "ANS-"  (A+B*A)
```

Another use of the direct mode is as an aid in program development and debugging. Through use of direct statements, program variables can be altered or read, and program flow may be directly controlled.

Direct statements in combination with program variables can be used in the following cases:

-after CTRL/C
-when an error occurs during program execution
-after a STOP-statement

This facility is not available at normal end of program.

2.2.3 Run Mode

A program consisting of a series of numbered statements can be executed only in the Run Mode. Execution (run mode) begins when the RUN command is entered: For example:

```
        80 A = 5®
        100 PRINT A®
        RUN®
        5
        *basic*
```

2.3 Line entry

Basic III is a "free format" language - the computer ignores extra blank spaces in a statement. For example, these four statements are equivalents:

```
        30  PRINT S
        30  PRINT   S
        30PRINTS
        30P RINT S
```

Basic III ignores spaces totally. A program is listed in a format that has no connection to the spaces you use to make your input more readable. It is important to note that spaces are significant in the following areas:

1. REM Statements
2. Extend Mode
3. Data Statements

2.3.1 Procedure

Lines input to Basic III are either executed immediately (Direct Mode) or stored in the user program area for later execution (Program Mode). Program mode statements can also be saved on disk for future execution. Basic III accepts lines when it is not executing a program.

The RETURN ® key must be pressed after each line.

Example:

```
        10  INPUT A,B,C,D,E®
        20  LET S=(A+B+C+D+E)/5®
        30  PRINT S®
        40  IF A <= 999 GOTO 10®
        50  END®
```

Pressing RETURN "®" informs Basic III that the line is complete. Basic III then checks the line for mistakes. If mistakes were made, an error message is displayed on the screen. (Refer to Section 2.4.)

2.3.2 Immediate Corrections

CTRL H acts as a backspace, deleting the immediately preceding character. (Note that CTRL X key deletes all preceding characters in a line.)

```
Typing these characters:       20 LR CTRL H ET S=10
is equivalent to typing:       20 LET S=10

And, typing these characters:  30 LET CTRL HHH PRINT S
is equivalent to typing:       30 PRINT S
```

The ED command (refer to Section 4) provides the facility to change characters after the line has been entered.

2.3.3 Deleting a Statement

To delete the statement being typed, depress CTRL X. This deletes the entire line being typed. For example:

```
        20 LET S = 12X CTRL X
```

To delete a previously typed statement, type the statement number followed by a RETURN "®". For example:

```
        5  LET S = 0
        10 INPUT A,B,C,D,E
        20 LET S = (A+B+C+D+E)/5
```

To delete Statement 5, above, type:

```
        5®
```

To delete blocks of statements, refer to ERASE, Section 4.

2.3.4 Changing a Statement

To change a previously typed statement (in program mode), retype it with the desired changes. The new statement replaces the old one.

To change statement 5 in the above sequence, type:

```
        5  LET S = 5®
```

The old statement is replaced by the new one. Use the LIST command to check what is left of the program.

If a single or a few characters need to be corrected, use the ED command. (Refer to Section 4.)

Blocks of statements from another program can be inserted by the MERGE commands. The user thereby has the possibility to handle programming on a modular basis.

2.4 Editing a program

Lines may be deleted, inserted or changed according to the procedures described previously in this section and the commands that are available in Basic III. The LOAD command places the desired program into working storage. The MERGE command allows you to combine or change your program with a set of statements loaded from a disk file. The ERASE command deletes blocks of statements. The ED command facilitates corrections of an existing line on a character basis.

When editing a program, you want to increase or decrease increments between selected lines. This is done by the RENUMBER Command after additions or deletions have been done.

If there is a syntax error in the previously typed statement, an error message is printed. This line is not entered but is retained in a save area and displayed on the screen with the cursor positioned at the character where the interpreter recognized the error. The erroneous line may be immediately edited using the same set of special keys as specified for the EDIT command. (Refer to Section 4.)

Example:

```
        10 PRONT CUR (2,30) "TROUBLE REPORT"®
        UNDECODEABLE STATEMENT
        10 PRONT CUR (2,30) "TROUBLE REPORT"
          -
          (Use CTRL-L until the cursor is positioned on the O in
          PRINT, enter I. Then press RETURN.)
        10 PRINT CUR(2,30) "TROUBLE REPORT"(R)
```

2.5 Executing a program

2.5.1 Start execution

The RUN command is provided to start the execution of a program. When the command (RUN ®) is entered, Basic III starts to execute the program in the user's program area at the lowest numbered line. Execution continues until either one of these conditions is encountered:

```
STOP
BYE
END
ERROR
```

or until the operator breaks by CTRL-C .

When the program executes a STOP or END statement it halts and all the variables are still in existence. The user can examine the variables by simply addressing the respective ones by the variable name. For example, you want to know the values of the variables A, S, and K%. Enter the following command:

```
PRINT A,S,K%®
```

The computer will then write the current values of the variables when program execution was stopped.

Errors cause an error message to be written on the screen. See Appendix B for the complete set of error messages.

2.5.2 Stop execution

CTRL-C	Stops a running program.
A stopped program:	CONT (or CON) Continue execution. CTRL-Z Single step the program. The program may be listed and variables examined before execution continues, but if any part of the program is changed (or even attempted to be changed) the program can not be continued.

2.6 Documenting a program

Basic III permits the programmer to document a program with notes, comments and messages. There are two methods available: Standard REM statements and text preceded by an exclamation point. The latter type of comments are easier to use since they can occur without a colon.

Examples:

```
a.  10 A = 7: REM ASSIGN "7" TO THE VARIABLE "A"
b.  10 A = 7 ! ASSIGN "7" TO THE VARIABLE "A"
```

REM lines are part of a BASIC program and are printed when the program is listed; however, they are ignored when the program is executing. Any series of characters may be used in a comment line. The remarks are usually marked with some clearly visible character, making them easily noticed in a program. For example:

```
        100 REM*** CAUTION ***
```

A comment cannot be terminated by a colon. The colon is treated as part of the remark. In the example below,

```
        150 REM ***INITIALIZER***:LET R1=3.5E2.1
```

the assignment statement will not be executed. The entire line is considered to be a non-executable comment.

Indentation is another method of documentation. The structure of hierarchy of a program can then be easily shown. This is done automatically for some BASIC statements, for example FOR ... NEXT.

2.7 Program communication

When several programs are executed in a multi-user environment, program communication may be through special drivers or through pipes. Refer to the description of the statement OPEN.

This should not be confused with the BASIC COMMON statement, which only keeps data in memory when a new program is loaded and started as the same task with the CHAIN statement.

2.8 File usage

Basic III facilities to define and manipulate input and output data on file structured devices like disk drives, as well as non-file structured devices, like console, printers and others.

Two methods are supported:

- Sequential (one record after another from the beginning of the file or random from a defined point in the file)

- Indexed Sequential (random by key). Indexed Sequential access is available with the ISAM option.

A data file consists of a sequence of data items transmitted between a BASIC program and an external input/output device. The external device can be the user's terminal, printer, disc or other device defined in the operating system.

Each data file is externally identified by a name, the file designator name (e.g. ABC123). Internally in the user's program, the file is accessed as a channel number. PREPARE, OPEN and CLOSE statements are used to establish and terminate a channel for the data transfer. All further references to the file in the program will be to the channel number (e.g. #1) not to file name - ABC123.

Random I/O permits the user's program to have complete control of I/O operations. Properly used, Random I/O is the most flexible and efficient technique of data transfer available under Basic III. It is, however, not as simple as Sequential I/O. Less experienced users should first experiment with the Sequential I/O techniques before attempting Random I/O. Random I/O is explained in detail in section 2.8.1 and 2.8.2.

The file number is defined in the program by means of one of the instructions PREPARE or OPEN. These statements will open the file, i.e. set up a channel for the data transfer. To close such a data transfer channel the instruction CLOSE is used. The instructions INPUT and PRINT or GET and PUT are used for the data transfer.

A buffer area is created by the system when a file is opened. All data transfer to and from a file is buffered.

Opening a File

To open an existing file the OPEN statement is used. If the file shall be created, it should be opened with a PREPARE statement.

Example:

```
        10 OPEN "mast1" AS FILE I
```

opens existing file named mast1 for input/output and assigns logical unit 1, for I/O, to that file.

Data Transfer To/From a File

The transfer of data takes place directly between the internal channel (the file number) and the string variable or the value of the expression in question. All data transfer refers to either a one-byte or one-character string (the characters followed by a carriage return). Using the line I/O statements INPUT, PRINT or INPUT LINE, the BASIC converts internal variables to strings and vice versa.

The following instructions can be used:

INPUT #	Reads values to variables or strings from the position of the file pointer to a line feed.
INPUT LINE #	Reads a value to a string variable from the position of the file pointer to a line feed. The line feed is replaced by a carrige return and line feed in the string variable. Also accepts embedded spaces and commas.
PRINT #	Writes the contents of variables into the file.
GET # COUNT nn	Reads one byte or the given number of bytes from the position of the file pointer.
PUT #	Writes a string into the file. In Record I/O mode, one record is written.

POSIT #	Moves the file pointer to the desired position.

If no file number is given in the GET statement, it will attempt to read from the keyboard. If the COUNT option is not used, GET will read one byte, i.e. one character.

Example:

```
        20 GET #1,D2$ COUNT 6%
```

will read file number 1 from the position of the file pointer six characters on. These characters are put in the string D2$.

For random access to a file, the instruction POSIT is used to position the file pointer at the given position in the file. The number of characters always refers to the beginning of the file (position 0). POSIT can be used together with any one of the other file handling instructions.

Example:

```
        LIST
        40 OPEN "pearl" AS FILE 1 ! "pearl" contains ABCDEFGHIJK.
        50 POSIT #1,5
        60 GET #1,A$ COUNT 3
        70 PRINT A$
        80 ; POSIT (1)
        90 END
        RUN
        FGH
        8
        *basic*
```

The function POSIT(<file number>) reads the position of the file pointer. In the example above, POSIT(1) returns the value 8, when the example has been executed. POSIT returns a floating point value, so that very long files can be handled.

Closing a File

The data transfer to or from a file will not be correctly terminated until the file is closed. The contents of the buffer area are then transferred.

There are two ways of closing a file:

CLOSE n[,n1,...]	closes the file(s) associated with file number n, n1,...
CLOSE	closes all files

Note! It is essential that files, which has been written to, is properly closed.

2.8.1 File creation

Basic III supports data files with the following type of record:

    Variable Length Records

That is, the size of the file subdivisions to which records correspond may be of variable length.

Only files with variable length records are available.

Variable Length Records

To allocate a data file of variable length records use the PREPARE statement. Data can be written, for example, using a PUT or PRINT statement as shown in the procedure below.

Examples:

```
PREPARE "filea" AS FILE 1       I PREPARE "fileb" AS FILE 2%
INPUT "NUMBER OF RECORDS?" R%   I
FOR I% = 1% to R%               I
  INPUT "ASCII DATA?" A$        I INPUT "NUMBER OF BYTES?" R%
  PRINT #1%,A$                  I FOR I%=1% TO R%
NEXT I%                         I   INPUT "BINARY DATA?" A$
                                I   PUT #2%, A$
                                I NEXT I%
```

Fixed Length Records

For creating an ISAM index file and its associated data file refer to the ISAM option description.

2.8.2 Access methods

Variable Length Records

Data files containing variable length records are accessed sequentially, with or without a random starting point, as shown in the following procedure: Note that the PREPARE statement always creates files with variable length records.

1. Specify OPEN statement with Byte I/O and READ/WRITE mode desired.

   Examples:
   ```
   OPEN "vol:filelist" AS FILE 1 !BYTE I/O
   OPEN "vol:filewrite" AS FILE 2 MODE 1% ! WRITE MODE
   ```

2. Sequential Access from the beginning of the file.

   The input/output is done from a point in the file, indicated by a file pointer. When a file is opened, the file pointer is automatically set to the beginning of the file. After an I/O operation the file-pointer is left on the next available character in the file. The POSIT statement may be used to set the file pointer to the desired position before I/O is performed.

2a. Use the INPUT or INPUT LINE statements to input lines of text from text files in ASCII format. Each line should be terminated by a 'LF' character (ASCII value 10 decimal). The maximum line length is 160 characters including two bytes for the 'CR' 'LF' characters.

The INPUT statement requires each data item in the line to be separated by a comma and ignores leading spaces input.

The INPUT LINE statement reads only to a string variable and also reads spaces, commas etc., but inserts the two characters 'CR', 'LF' (ASCII 13,10) at the end of the string.

Examples:
```
    INPUT #1,A,B$,C%
    INPUT LINE #1,C$
```

When reading from the console device only, the characters are echoed.

2b. The GET statement is used to input 8-bit binary data. The number of characters to input must be given, unless only one character shall be input. If COUNT is omitted in the statement, only one byte is read.

Examples:
```
    GET #1,A$ COUNT 289
```

Note:
GET without file number reads binary data from the terminal. This data is 1 bit binary data (i.e. the most significant bit is always 0).

2c. The PRINT statement are used to output lines of ASCII text, followed by a line feed character 'LF', unless the ';' is given at the end of the PRINT statement.

Examples:
```
    PRINT #1, "STRING=";A$
or
    PRINT #1, A$;
    PRINT #1, B$         to build a line with two statements.
```

The normal file handling assumes that PRINTing is done sequentially and that the file ends after the last printed line.

2d. The PUT statement is used to output binary data. An entire string are output to the file as a string of 8-bit data.

Example:
```
    PUT #1,A$
```

It is essential that a file, which has been written to, is properly closed.

3. Random access are done by defining a starting point with the POSIT statement, from which data is read sequentially.

The POSIT statement sets the file pointer for the subsequent GET, INPUT, INPUT LINE, PUT or PRINT statements.

Example: Read 10 bytes from byte number 235 in the file.

```
    POSIT #1,235
    GET #1,A$ COUNT 10
```

2.9 Logical units

Basic III ensures independence from physical input/output devices through the use of file numbers. The file number can be treated as a logical unit and is handled with the instructions OPEN, PREPARE and CLOSE. The file number may, for instance, represent a printer or a file on a disc.

Example:

```
    10 - -
    20 OPEN "PR:" AS FILE 2 ! Open the printer
    30 - -
    40 - -
    50 CLOSE 2 ! Close the printer
    60 END
```

Note: Refer to section 4 for information about colon expression replacement.

2.10 Error handling

Certain errors can be detected by Basic III when it executes a program. These errors can, for instance, be computational errors (such as division by 0) or input/output errors (reading and end-of-file code as the input to an INPUT statement). Normally, the occurrence of any of these errors will cause termination of program execution and the printing of a diagnostic message or an error number, depending on the presence of the basicerr.txt file. Compare appendix B.

Some applications may require that program execution continues after an error has occurred. To accomplish this, the user can include an ON ERROR GOTO <line number> statement in the program.

The ON ERROR GOTO statement should be placed before all the executable statements with which the error handling routine deals.

When an error occurs in a program, Basic III checks to see if the program has executed an ON ERROR GOTO statement. If not, a message is printed at the screen and the program execution is terminated.

If an ON ERROR GOTO statement has been executed, the program execution will continue at the line number specified by that statement. The subroutine at that line number can test the function ERRCODE to find out precisely what error has occurred and decide what action is to be taken. The exit from the error routine may be to the statement, causing the error, or to another statement in the same program segment.

If there are portions of the program in which any errors detected are to be processed by the system and not by the subroutines of the program, the error subroutine can be disabled by executing the ON ERROR GOTO statement without the line number reference.

```
    line number   ON ERROR GOTO
```

The computer will then attend to all errors as it would do if no ON ERROR GOTO <line number> had ever been executed.

One of two types of error handling sequences are possible, regarding how to exit from the error routines. The system searches before start, for any RESUME statement in the program code. If any RESUME statement is found, the standard error sequence is selected, otherwise the optional sequence is used.

Standard error sequence:

Exit from an error routine, for continuous execution, MUST be with a 'RESUME' or 'RESUME line.no' statement, to restore the error condition.

Standard error handling is according to the following:

1. ON ERROR GOTO line.no sets the 'Error Trap' flag and defines the error routine entry line.no.

2. When an error is detected, the error routine is executed under the following conditions:

   - The 'Error Trap' flag is reset, defining that all errors shall cause a system error message and termination of the program. The only way to re-enable a new user error routine, is to exit with a RESUME statement.

   - Any 'ON ERROR GOTO line.no' statement may define a new user error routine entry point.
     NOTE, however, that this will be activated first after the exit from the present error routine.

   - A blank 'ON ERROR GOTO' statement can be used to exit from the error routine, but will terminate the program with a system error message.

   - The 'RESUME' or 'RESUME line.no' statement defines the exit from an error routine, restoring the error conditions for continuous execution.

3. The 'RESUME' statement restores the 'Error Trap' and returns to the statement, where the error occurred. The defined user error routine is enabled.

   - If continuous execution is not possible, the program is terminated.

   - If the error was an I/O error, the statement is executed again from the beginning.

- If the error was not an I/O error, the return is to a point after the operation, causing the error, and the rest of the statement is executed.

  Example 1: I/O error

```
LIST
10 ON ERROR GOTO 50
20 INPUT "X=" X
30 ; X
40 END
50 ; : RESUME
RUN
X=ABC           <error causes a new display of 'X='>
X=56
56
*basic*
```

  Example 2: Calculation error. The result of the division with zero will be set to the highest possible value, approx. 1.7E+38, before the rest of the statement is executed.

```
LIST
10 ON ERROR GOTO 50
20 X=5/0/1.7E+38
30 PRINT X
40 END
50 RESUME
RUN
1.00083
*basic*
```

4. The 'RESUME line.no' statement does the same as the 'RESUME' statement, but the return is not to the erroneous statement.

   NOTE! that the return must be to a statement at the same subroutine or function level as the erroneous statement, to continue the program execution with the correct system stack pointers.

Optional error handling sequence:

If no RESUME statement exists in the program, no automatic return to the erroneous statement is available. The exit from an error routine is defined by the execution of a new 'ON ERROR GOTO line.no' or 'ON ERROR GOTO' statement, which also resets the error conditions.

Note that also in this case, the error routine must return to the same subroutine or function level, where the error occurred.

Example of error handling:

```
        LIST 10-120
        10  ON ERROR GOTO 100 !At erroneous input go to line 100
        20  INPUT "AGE, WEIGHT " A,W
        30  ON ERROR GOTO 10 !Disable the error handler
```

```
40  STOP
100  PRINT !Error handler
110  PRINT " Erroneous input! "
120  GOTO 20 !Jump to line 20
*basic*
```

3. CONVENTIONS AND SYNTAX

Contents

3. CONVENTIONS AND SYNTAX

3.1 Program conventions

3.1.1 Name structure

A user program is composed of one or more properly formed Basic III statements, constructed with the language elements and syntax described in the following sections. A statement contains instructions to Basic III. A program line begins with a line number followed by one or more Basic III statements, up to a maximum of 160 characters. Line numbers indicate the particular sequence of execution. Each statement begins with a keyword specifying the type of operation to be performed. A program line can also contain multiple statements.

Each statement gives an instruction to the computer (in this example PRINT):

```
        30 PRINT S
```

The value currently assigned to the variable "S", above, is printed. If the instruction requires further details, operands (numeric details) are supplied. The operands specify what the instruction acts upon, (for example, GOTO):

```
        40 GOTO 10
```

In the above example, the operand "10" is the line number to which program control will be transferred upon execution of the "GO TO" statement.

The last statement in a program, as shown here, is an END statement.

```
        10 INPUT A,B,C,D,E
        20 LET S = (A+B+C+D+E)/5
        30 IF A=999 GOTO 60
        40 PRINT S
        50 GOTO 10
        60 END
```

The END statement informs the computer that the program is finished, but its presence is not mandatory.

3.1.2 Line numbering

Each program line in the program mode is preceded by a line number. A line number has the following effects.

1. Indicates the order in which the statements are executed. The statements may be written in any order.

2. Enables the normal order of evaluation to be changed by GOTO, GOSUB statements, etc.

3. Permits program modification of any specified line without affecting any other portion of the program.

The line number is chosen by the programmer. It may be any integer from 1 to 65,535 inclusive. The system uses the line numbers to keep the program lines in order and for the execution required.

Program lines may be entered in any order; they are usually numbered by fives or tens so that additional statements can be easily inserted. The computer keeps them in numerical order no matter how they are entered. For example, if the program lines are input in the sequence 30, 10, 20, Basic III rearranges them in order: 10, 20, 30. There are commands for automatic line numbering (AUTO) and for renumbering (REN).

3.1.3 Statements

A program line begins with a line number followed by a Basic III statement. The keyword of a Basic III statement identifies the type of statement. Basic III is thereby informed what operation to perform and how to treat the data - if any - that follows the keyword.

Multiple Statements on a Program Line

The user is allowed to write more than one statement on a single line. Each multi-statement (except the last) is terminated with a colon. Only the first statement on the program line can have a line number preceding it.

Example:

```
        100 PRINT A,B,C
          is a single statement program line.
        200 LET X=X+1 : PRINT X : IF Y = 1 GOTO 100
          is a multiple statement program line containing three
          statements: LET, PRINT and IF-GOTO.
```

As a rule any statement can be used anywhere in a multiple statement line. The exceptions to the rule have been explicitly specified in individual statement descriptions.

3.1.4 Expressions

Expressions are a fundamental building block used in many Basic III statements. The primary elements of expressions are constants, variables, arrays and functions. These elements are then combined using arithmetic, relational and/or logical operators, to form expressions. This and succeeding sections will define these terms within the context of Basic III.

Arithmetic expressions

An arithmetic expression has an arithmetic value which is either floating point or integer. Mixed expressions (i.e., both floating point and integer) yield a floating point value. The following mathematical operators can be used in arithmetic expressions:

Operator	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division
^ or **	Exponentiation (using 'up-arrow' or two stars)
- (unary)	Subtraction or negation

No two mathematical operators may appear in sequence and no operator is ever assumed (e.g., A++B and (A+2) (B-3) are not valid).

Examples of Arithmetic expressions:

```
    4.123
    3 + A
    A% +50
    B * (C**3 + 1.5)
    PI *R**2
```

Relational expressions

A relational expression yields a truth value that reflects the result of comparing two values. Symbolically it can be defined as:

    <expression><relational operator><expression>

Expression can be either an arithmetic term or a string term but not both in a single expression.

The relational symbols Basic III allows are:

Mathematical Symbol	Basic III Symbol	Example	Meaning
=	=	A=B	A is set equal to B
<	<	A<B	A is less than B
>	>	A>B	A is greater than B
≤	<=	A<=B	A is less than or equal to B
≥	>=	A>=B	A is greater than or equal to B
≠	<>	A<>B	A is not equal to B

Example:

```
    X>Y
    NUM8<=0
    A=B
```

Examples of string relational symbols are shown in Section 3.3.

Logical expressions

A logical expression yields a truth value that reflects the existence or nonexistence of a particular condition.

A logical expression is one of the following:

1. An integer expression (FALSE if 0, TRUE if <> 0).
2. A set of relational expressions, corrected by logical operators.
3. A set of integer expressions, or logical expressions, or both, connected by logical operators.

Logical operators are used in IF - THEN and such statements where some condition is used to determine subsequent operations within user program.

The logical operators are as follows (where A and B are RELATIONAL EXPRESSIONS):

Basic III Operator	Example	Meaning
NOT	NOT A	The logical negative of A. IF A is true, NOT A is false.
OR	A OR B	A OR B has the value true if either or both A or B are true and has the value false only if both A and B are false.
XOR	A XOR B	The logical exclusive OR of A and B. A XOR B is true if either A or B is true but not both, and false otherwise.
IMP	A IMP B	The logical implication of A and B. A IMP B is false if and only if A is true and B is false; otherwise the value is true.
EQV	A EQV B	A is logically equivalent to B. A EQV B has the value true if A and B are both true or both false, and has the value false otherwise.
AND	A AND B	The logical product of A and A. A AND B has the value true only if A and B are both true and has the value false if A or B is false.

3.1.5 Variables

A variable is a data item whose value can be changed during program execution. A numeric variable is denoted by a fixed variable name.

Two modes dictate the length of a variable name: EXTEND and NO EXTEND.

In EXTEND mode variable names of up to 32 characters are permitted, but spaces are required to delineate names and functions unless the adjoining characters is a line number or arithmetic operator. In NO EXTEND mode variable names of one letter and an optional digit is allowed but spaces are unnecessary. The default is NO EXTEND mode. The following are the letters and digits which can be used to form variable names: A,B,...,Z and 0,1,...,9.

A name can also have an FN prefix (denoting a function name), a % suffix (denoting an integer), a . suffix (denoting floating point), a $ suffix (denoting a string), or a subscript suffix that consists of a set of subscripts enclosed in parentheses.

A string expression is a value that consists of a sequence of characters, each character occupying a byte. A string expression can be expressed either as a sequence of characters enclosed in quotation marks or as a variable by a variable name with a $ suffix.

Mixing of data types in a statement should be avoided. Use integers whenever possible.

The same name in combination with various prefixes and suffixes can appear in the same program and generate mutually independent variables. For example, the name A refers to a floating point variable A. The name A can be used as follows:

A	floating point variable A
A%	integer variable A%
A$	string variable A$
A(d)	floating point array A with dimension subscript d
A%(d)	integer array A% with dimension subscript d
A$(d)	string array A$ with dimension subscript d
FNA	floating point function A
FNA%	integer function A%
FNA$	string function A$

In the EXTEND mode a name can be used as follows:

SECANT	floating point variable SECANT
SECANT%	integer variable SECANT%
SECANT$	string variable SECANT$
SECANT(d)	floating point array SECANT with subscript d
SECANT%(d)	integer array SECANT% with subscript d
SECANT$(d)	string array SECANT$ with subscript d
FNSECANT	floating point function SECANT
FNSECANT%	integer function SECANT%
FNSECANT$	string function SECANT$

Variables are assigned values by LET, INPUT and READ among other statements. Variables are set to zero before program execution.

It is necessary to assign a value to a variable only when an initial value other than zero is required. To ensure that later changes or additions will not cause problems it is good programming practice to always initialize all variables to zero.

Subscripted variables (array) and the DIM statement

In addition to the simple variables the use of subscripted variables (arrays) is allowed. Subscripted variables provide the programmer with additional computing capabilities for dealing with lists, tables, matrices, or any set of related variables. Variables are allowed one (vector) or two or more (matrix) subscripts.

The name of a subscripted variable is any acceptable variable name followed by one or two integers enclosed in parentheses. For example, a list might be described as A(I) where I goes from 0 to 5 as follows: A(0), A(1), A(2), A(3), A(4), A(5).

This allows the programmer to reference each of six elements in the list, which can be considered a 1-dimensional algebraic vector as follows:

```
A(0)
A(1)
A(2)
A(3)
A(4)
A(5)
```

A 2-dimensional matrix B(I,J) can be defined in a similar manner. It is graphically illustrated below:

B(0,0)	B(0,1)	B(0,2)	B(0,3)	...	B(0,J)
B(1,0)	B(1,1)	B(1,2)	B(1,3)	...	B(1,J)
B(2,0)	B(2,1)	B(2,2)	B(2,3)	...	B(2,J)
B(3,0)	B(3,1)	B(3,2)	B(3,3)	...	B(3,J)
..	..	..	..	...	..
..	..	..	..	...	..
B(I,0)	B(I,1)	B(I,2)	B(I,3)	...	B(I,J)

Subscripts used with subscripted variables can only be integer values. Subscripts are truncated to integers if they are of floating type.

A (DIM) dimension statement is used to define the maximum number of elements in an array.

Arrays may start with subscript 0 or 1. An array dimensioned A (5), will have 5 elements if option base 1 is specified or 6 elements if option base 0 is specified. The default is option base 0. If an option base is specified, it must be declared before any array is dimensioned or used.

If a subscripted variable is used without a DIM statement, it is assumed to be dimensioned to length 9 or 10 in each dimension (that is, having 10 or 11 elements in each dimension, 1 through 10 or 0 through 10 respectively). DIM statements are usually grouped together among the first lines of a program.

The first element of every matrix is automatically assumed to have a subscript of (0,0), if OPTION BASE 1 is not specified.

Example: OPTION BASE 0

```
10  REM - MATRIX CHECK PROGRAM
20  DIM A(4,8)
30  FOR I=0 TO 4
40  LET A(I,0)=I
50  FOR J=0 TO 8
60  LET A(0,J)=J
70  PRINT A(I,J);
80  NEXT J
90  PRINT
100 NEXT I
999 END

RUN
0   1   2   3   4   5   6   7   8
1   0   0   0   0   0   0   0   0
2   0   0   0   0   0   0   0   0
3   0   0   0   0   0   0   0   0
4   0   0   0   0   0   0   0   0
```

Example: OPTION BASE 1

```
10  REM - MATRIX CHECK PROGRAM
15  OPTION BASE 1
20  DIM A(4,8)
30  FOR I=1 TO 4
40  LET A(I,1)=I
50  FOR J=1 TO 8
60  LET A(1,J)=J
70  PRINT A(I,J);
80  NEXT J
90  PRINT
100 NEXT I
999 END

RUN

1   2   3   4   5   6   7   8
2   0   0   0   0   0   0   0
3   0   0   0   0   0   0   0
4   0   0   0   0   0   0   0
```

Notice that a matrix element, like a simple variable, has a value of 0 until it is assigned a value.

3.1.6 Constants

Numeric constants retain a constant value throughout a program. They can be positive or negative. Numeric constants can be written using decimal notation as follows:

```
+3
-4.567
12345.6
-.0001
-1.234E+5
```

The example constants would be stored as floating point, since they have no % suffix.

The use of an explicit decimal point or percent sign is recommended in all numeric constants to avoid unnecessary data conversions and to improve documentation.

3.1.7 Reserved words

ABS	EXTEND	NEXT	SCR
ADD	FIELD	NO	SGN
ASCII	FIX	NOTRACE	SHORT INT
ATN	FLOAT	NUM	SIN
AUTO	FN	OCT	SINGLE
BYE	FNEND	ON	SLEEP
CALL	FOR	OPEN	SQR
CHAIN	GET	OPTION BASE	STAT
CHR	GOSUB	OPTION EUROPE	STATUS
CLEAR	GOTO	OR	STOP
CLOSE	HEX	OUT	STRING
CLS	IF	PEEK	SUB
COMMON	IFEND, ELSE	PEEK2	SWAP
COMP	INP	PEEK4	SWAP2
CONTINUE	INPUT	PI	SYS
COS	INSTR	POKE	TAB
CUR	INT	POSIT	TAN
CVT	INTEGER	PREPARE	THEN
CVTF	KILL	PRINT	TIME
DATA	LEFT	PUT	TRACE
DEF	LEN	RANDOMIZE	UNSAVE
DIGITS	LET	READ	UNTIL
DIM	LIST	REM	USING
DIV	LOAD	REN	VAL
DOUBLE	LOG	RENUMBER	VAROOT
EDIT	LOG10	REPEAT	VARPTR
ELIF	LONG INT	RESTORE	WEND
ELSE	MERGE	RESUME	WHILE
END	MID	RETURN	
ERASE	MOD	RIGHT	
ERRCODE	MUL	RND	
ERROR	NAME	RUN	
EXP	NEW	SAVE	

3.2 Integer and floating point

Normally, all numeric values (variables and constants) specified in a Basic III program are stored internally as floating point numbers. If the numbers to be dealt with in a program are intergers, significant economies in storage space can be achieved by use of the integer data type. Integer arithmetic is also faster than floating point arithmetic. This section discusses integer and floating point operations within the context of Basic III. Higher precision may be obtained, using the string arithmetic functions with numeric values as strings with max. 126 digits length. Refer to section 3.3.

A constant, variable or function can be specified as an integer by ending its name with the % character.

Example:

```
        A%   FNX% (y)
        -8%  Z3%
```

The user always has to specify with a %-character to indicate where an integer is to be generated. Otherwise, a floating point value is produced. The opposite holds when the non-default INTEGER mode has been selected. In the INTEGER mode, all variables etc. are considered as integers if not marked with a decimal point after the name.

```
Example in INTEGER mode:     A    FNX(y)      integers
                             -8.  Z3.         floating point
```

When raising to an integer power, the power value should be indicated explicitly as an integer.

Basic III automatically converts integers and floating point variables to the desired format, required as argument to Basic III statements or functions.

Floating Point Values

Floating point values range from:

$2.93874 \times 10_{-39}$ through $1.70141 \times 10_{38}$ - single precision

1.79769313486232E+308 through
4.4501477170144E-308 - double precision

All floating point variables and expressions are calculated to single or double precision. Mixing of precision is not possible. The default is single precision.

Single precision allows for six digits of significance and double precision allows for sixteen digits. Numbers are internally rounded, using 5/4 round method to fit the appropriate precision. Numbers may be entered and displayed in three formats:

1. Whole - 153
2. Fractional - 34.52
3. Scientific Notation (E-format) - 136E-2

Integer Values

The range of integer numbers is:
    -2147483648 through 2147483647

3.2.1 Use of integers as logical variables

Integer variables or integer valued expressions can be used within IF statements in any place that a logical expression can appear. Any non-zero value is defined to be true and an integer value of 0% corresponds to the logical value false. The logical operators (AND, OR, NOT, XOR, IMP, EQV) operate on logical (or integer) data in a bit-wise manner.

Note: The integer -1% is normally used by the system when a true value is required. Logical values generated by Basic III always have the values -1% (true) and 0% (false).

Logical operations on integer data

Basic III permits a user program to combine integer variables or integer valued expressions using a logical operator to give a bit-wise result.

For the purpose of logical operations the truth tables following are valid. A is the condition of one bit in one integer value and B is the condition of the bit in the corresponding bit position of another integer value.

The truth tables are as follows:

A	B	A AND B	A OR B	A XOR B	A EQV B	A IMP B	NOT A
1	1	1	1	0	1	1	0
1	0	0	1	1	0	0	0
0	1	0	1	1	0	1	1
0	0	0	0	0	1	1	1

The result of a logical operation is an integer value generated by combining the corresponding bits of two integer values according to the rules shown above.

The result of any logical operation can be assigned to an integer or a floating point variable.

Example:
```
        10  REM BIT VALUES: 13 = 00001101, 14 = 00001110
        20  REM ** 13 or 14 = 00001111 = 15
        30  A% = 13% OR 14%
        40  PRINT A%
        RUN
        15
        *basic*
```

Variables and valued expressions can be operated on by AND, OR, XOR, EQV, IMP and NOT to give a bit-wise integer result. If logical operations are done on floating point variables or floating point valued expressions, conversion to integer format is done before the execution of the logical operation.

Example:
```
        100 IF A% AND 1% THEN
    is the same as:
        100 IF A% AND 1.6 THEN
```

3.2.2 Input/Output with integers and floating point

Input and output of integer variables is performed in exactly the same manner as the corresponding operations on floating point variables.

Any number which can be represented by up to six significant digits in single precision mode (or 15 digits in DOUBLE precision mode) is printed without using the exponential form.

Any floating point variable that has an integer value is automatically printed as an integer but is internally still a floating point number.

If more than six digits (single precision) or fifteen digits (double precision) are generated during any computation using floating point numbers, the result is automatically printed in E-format:

```
        [-] x Ey
        where: - = Sign of the number, if number is negative
        x = A maximum of six digits for single precision and
            fifteen digits for double precision
        E = Represents the expression "times 10 to the power of"
        y = An exponent in the range (-38 through +38) single
                                     (-308 through +308) double
```

Examples:

5E-06 = $5 \times 10_{-6}$ = .000005

-125E+4 = $-125 \times 10_{4}$ = -1250000

Input allows all the formats used in output. When a floating point value is assigned to an integer variable, then the fractional portion is lost and rounded off to an integer.

3.2.3 Integer arithmetic

Arithmetic with integer variables is performed in modulo 2**32. The number is -2147483648 to +2147483647 and can be regarded as a continuous circle with -2147483648 following +2147483647.

Integer division forces truncation of any remainder. Note that the function MOD makes the remainder available.

Example:

```
        3%/4% = 0 and 283%/100% = 2.
```

When an operation is performed with both integer and floating point data, the operation is executed with floating point precision but the result is stored in the format indicated as the resulting variable.

Example:

```
        10    LET B% = Z% + 3/X
```

The result is truncated to give B% an integer value.

Mathematical operations

When more than one operation is to be performed in a single formula, rules are observed as to the precedence of the operators. The arithmetic operations are performed in the following sequence. Operation described in item 1 has precedence.

1. Any formula within parentheses is evaluated first. Then the parenthesized quantity is used in further computations. Where parentheses are nested the innermost parenthetical quantity is calculated first. For example, (A+(B*(C**3))) is evaluated as follows:

   Step 1 - (C**3), Step 2 - (B*(C**3)), and Step 3 - (A+(B*(C**3))).

2. In absence of parentheses the following precedence is performed:

   a. Intrinsic or user-defined functions
   b. Exponentiation (**)
   c. Unary, minus (-), that is, a negative number or variable
   d. Multiplication and division (* and /)
   e. Addition and subtraction (+ and -)
   f. Relational operators (=, <>, >=, <, <=, >)
   g. NOT
   h. AND
   i. OR and XOR
   j. IMP
   k. EQV

   Thus, for example, -A**B with a unary minus, is a legal expression and is the same as -(A**B). This implies that -2**3 evaluates as -8. The term A**-B is not allowed; however, A**(-B) is allowed.

3. In absence of parentheses, operations on the same level are performed left to right, in the order the formula is written.

User defined integer functions

An user function is defined to be of integer type by including the "%" suffix following the function name.

Example:
```
        10  DEF FNA%(X%) = X% * (Z% + X%)
```

A floating point function could be written as:

Example:
```
        10  DEF FNV (X%)=X%*(Z+X%)
```

3.2.4 Representation of numeric data in Basic III

There are three possible way to represent numeric data: INTEGER, STRING or FLOATING POINT. Each representation has its advantages and disadvantages. The choice of representation depends on:

1. The range of values of the data
2. The memory space available for the data storage
3. The required computation speed
4. The required accuracy of computation

INTEGER data

INTEGER data is	- exact
	- fast
	- requires only 4 bytes of storage
	- limited in range to 9 1/2 digits
	- requires awareness of overflow possibility

Integer data, variables, constants and arrays are stored in four bytes, i.e. 32 bits with two's complement binary notation. The value range is limited to -2147483648 to +2147483647.

Add, subtract and multiply operations produce exect result, but be careful of overflow, as Basic does not trap overflow as long as the result is within the 32 bit un-signed range (0 - 4294967295). An integer divide produces the truncated quotient Q=A/B and the remainder of A/B may be obtained by R=A-Q*B.

STRING data

STRING data is	- exact within the selected range
	- slow (1/2 the speed of floating point depending on accuracy selected)
	- requires large memory space, one byte/digit plus sign and decimal point.
	- large range, selectable max 126 digits

Strings containing legitimate numeric values can be manipulated with built-in functions. The string has the same form as a numeric constant.

The ADD$ and SUB$ functions provides explicit control over the number or decimal places in the result. The MUL$ and DIV$

functions allow either decimal place control or precision control:

ADD$(A$,B$,n) SUB$(A$,B$,n)	yields a result string with n decimal places.
MUL$(A$,B$,n) DIV$(A$,B$,n)	yields a result string with n decimal places if n is positive, but with '-n' digits of precision if n is negative.
VAL(A$)	converts a numeric string to a floating point number.
NUM$(F)	converts a floating point number to a string with the number of digits according to the earlier given DIGITS statement.
COMP%(A$,B$)	compares the algebraic values of two strings.

String numeric data provide the largest range of any data type. Up to 126 digits (including sign and decimal point) can be handled. In addition all computation is exact within the programmer defined limits of decimal places and/or precision.

The disadvantage of the string variables is their size. One byte for each digit, plus two for possible sign and decimal point plus the normal overhead tables common to all string variables. Also a string arithmetic statement requires more program space than an equivalent integer or floating point statement.

The string result of a string arithmetic function is always left justified. The length of the result string varies with the result.

Floating point data

FLOATING POINT is
- inexact
- larger significance:
  7 digits in SINGLE prec.mode
  15 digits in DOUBLE prec.mode
- larger range 2.9E-39 to 1.7E+38 Single
  ±IE±308 Double
- 4 bytes in SINGLE precision mode
  8 bytes in DOUBLE precision mode
- slow, 3 - 5 times slower than INTEGER

It is not possible to mix precision modes in a program.

Care must be taken when using floating point variables when comparing two numbers, due to:

- floating point calculations are by their nature inexact.

- since floating point numbers are internally stored as binary numbers, with a binary exponent, conversion between decimal and binary must be performed whenever a decimal number is moved to or from a variable, like when a variable is printed or input.

The inexactness is due to the fact that only 7 or 15 digits (approx) is retained in any computation. As an example, adding two nearly equal numbers with opposite signs introduces insignificant figures. As a consequence, (A+B)+C in a floating point computation is NOT always equal to A+(B+C).

```
Example:   A =  2222223
           B = -2222221
           C =  1.544444

then     A+B =  2.000000
     (A+B)+C =  3.544444

but      B+C = -2222219
     A+(B+C) =  4.000000
```

The binary decimal conversion problem is mainly due to that decimal fractions are not always exactly representable as binary fractions. In particular the decimal fraction 0.1 has the binary expansion 0.00011 0011 0011 ..... (an infinite expansion). Converting the 0.1 (decimal) to floating point must be either rounded or truncated, which introduces a small error. This can become magnified, depending on the calculations performed and show up in a significant digit later.

One possible approach when the accuracy must be kept high, is to bias the variables so that they always take integer values. As an example currency should be expressed in the smallest coin (Swedish crowns should be expressed in 'oere', i.e. 1 SEK = 100 oere). This approach requires the programmer to remember any scale factors used and adjust the computations accordingly. The FIX function can be used if necessary to keep variables as pure integers and perform rounding.

Example:

FIX(A*0.5 + .5) calculates 50% of A, but still rounded to a whole number.

If the programmer decides to retain variables with their natural radix point the DIGITS statement can be used to specify the number of decimal digits obtained in binary to decimal conversions. This specification applies to the PRINT and the NUM$ statements. The DIGITS statement is dynamic, so that various precisions can be used in the program.

Comparing floating point numbers, whose values may not be exact, can be a problem. There are three ways to overcome this:

1 - Use the FIX function to limit the number of decimal places.
```
IF FIX(A*100) = 0 ......
```

2 - Use the NUM$ and COMP% functions to convert the floating point number, according to the DIGITS precision and compare the items as string variables.
```
COMP%(NUM$(A),"0") = 0 .....
```

3 - Compare the values, but define a small difference to mean that the values are equal.
```
IF (A > -.001) AND (A < 0.001) ......
```

Physical representation of floating point numbers

Every floating point number is represented by the exponent part (e) and the fraction part (f), such as

$$\text{Number} = [-]\ f * 2_e \qquad 1/2 <= f < 1$$

Example: the number 1.5 is represented by $0.75 * 2_1$

The single, double representation follows the IEEE standard for floating point.

Single:
```
        <1> <--8--> <---23--->
         s     e         f
```

The first byte contains the sign of the fraction and the 7 most significant bits of the exponent.

The least significant bit of the exponent is stored in bit 7 of the second byte.

The exponent is biased by 128 to avoid using any sign in the exponent.

Double:
```
     <1> <--11--> <--52-->
      s      e        f
```

The remaining 23 or 52 bits hold the fraction as a positive binary fraction in the range 1/2 <= f < 1. This value is said to be normalized, i.e. no leading binary zeroes are allowed. As all numbers are normalized, the first bit is always 1, which is used in Basic III, to achieve one bit more in precision. The first bit is never stored in the memory and the second bit is stored in bit 6 or bit 3 (Double) of the second byte etc. to achieve 24 bits of accuracy in the fraction. An example is the decimal number 0.5, which can be exactly represented by:

exponent	= 0	(stored in first byte as 128 dec.)
sign	= 0	
fraction	= 0	(Hidden bit gives number = $1 * 2_{-1}$)

3.3 Strings

Besides the manipulation of numerical information Basic III also processes information in the form of character strings. A character string is a sequence of characters. This section defines string elements within the context of Basic III.

3.3.1 String constants

Character string constants are allowed analogous to numerical constants. Character string constants are delimited by either single (') or double (") quotes. If the delimiting character occurs twice in a string sequence it is considered as part of the text constant.

The value LET'S can be expressed in two ways: "LET'S" or 'LET''S'.

Examples:
```
        10  A1$ = "CHARLES"
        20  IF A$ = "GOOD" GOTO 40
        30  B$ = 'DON''T'             (has the value DON'T)
```

3.3.2 String variables

Any legal name followed by a dollar sign ($) character is a legal name for string variable.

Examples:
```
        A$,B4$ are simple string variables.
        B$(8),H5$(N,O),J$(K) are subscripted string variables.
        AMOUNT$(4) - (EXTEND MODE ONLY)
```

Note: The same name, without the $, denotes a numeric variable which can be used in the same program.

The same name can be used as a numeric variable and as a string variable in the same program.

Example:
```
        A,A$ and A% are allowed in the same program.
```

3.3.3 Subscripted string variables

The DIM-statement is used to define string lists and string matrices.

Examples:
```
        DIM W$(2,4)=8 !STRING LENGTH 8 maximum subscript values
                       2 and 4
        DIM R5$(9,9) !STRING LENGTH UP TO 80; maximum subscript
                       values 9 and 9
        DIM NAME$(7,6,3,2)=10 !STRING LENGTH 10; four-
                       dimensional matrix with maximum subscript
                       values 7,6,3 and 2
```

3.3.4 String size

The length of a non-dimensioned string variable is automatically set to the current length the first time the string is assigned a non-null value (<>'').

If less than 80 characters are used, then a default length of 80 characters is assigned.

Each string, both scalar and each array element, has two lengths:

1. Max length is the number of bytes allocated to the string.

2. Current length is the number of bytes currently in use. Current length may vary between zero and max length. The current length is the only visible length; this length may be examined by the function LEN etc.

Both lengths are initiated to zero as a program is started. They are modified when the string is dimensioned or assigned. If a string is assigned a null value (="") the current length will be set to zero. No further action is taken.

If a string is assigned a non-null value and has a non-zero max length, the string length is checked. If the string length is sufficient, a number of bytes will be allocated to store the data and the current length will be set to the number of allocated bytes. If the string length is not sufficient, an error message will be written. A maximum of 80 bytes is always allocated if DIM is not used.

3.3.5 String functions

Basic III provides various functions for use with character strings. These functions permit the program to:

.perform arithmetic operations with numeric strings

.concatenate two strings

.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 large string etc.

Section 6.2 discusses each string function in detail.

3.3.6 String arithmetic

The string arithmetic features functions that treat numeric strings in arithmetic operands. This is a way to perform calculations with greater precision. Numeric string variable names must be suffixed with a dollar sign ($) character. Numeric string constants must be bounded by quotation marks (") or apostrophes (').

The maximum size of a string arithmetic operand is 126 characters including the sign and the decimal point.

3.3.7 String input

The READ, DATA and INPUT statements can also be used to assign data to string variables in a program.

Example:
```
        10  INPUT "YOUR ADDRESS?";A$,
        20  INPUT "YOUR NAME?";B$
```
is the same as
```
        10  PRINT "YOUR ADDRESS";
        20  INPUT A$
        30  PRINT "YOUR NAME";
        40  INPUT B$
```

INPUT LINE is useful for string input because it accepts embedded blanks, commas, etc. It accepts only one line from the keyboard including carriage return and line feed.

Example:
```
        10  INPUT LINE D$
```

Example:
```
        10  READ A, B, C$, D
        20  DATA 17, 14, 61, 4
```

This results in the following assignments:
```
        A  = 17
        B  = 14
        C$ = character string "61"
        D  = 4
```

The INPUT statement is used to input character strings exactly as though accepting numeric values. String constants are not allowed in string input statements.

3.3.8 String output

Only those characters that are within quotes are printed when character string constants are included in PRINT statements. The delimiters are not printed:

Example:
```
        10 PRINT "ALL IS OKAY"
        RUN
        ALL IS OKAY
        *basic*
```

Strings can also be stored in files on an output device.

3.3.9 Relational operators

The relational operators, when applied to string operands, indicate alphabetic sequence.

Example:
```
        15  IF  A$(I)<A$(I+1)   GOTO 115
```

When line 15 is executed the following occurs: A$(I) and A$(I+1) are compared; if A$(I) occurs earlier in alphabetical order than A$(I+1), execution continues at line 115.

The chart below contains a list of the relational operators and their string interpretations.

Operator	Example	Meaning
=	A$=B$	The strings A$ and B$ are equivalent.
<	A$<B$	The string A$ occurs before B$ in collating sequence.
<=	A$<=B$	The string A$ is equivalent to or occurs before B$ in collating sequence.
>	A$>B$	The string A$ occurs after B$ in collating sequence.
>=	A$>=B$	The string A$ is equivalent to or occurs after B$ in collating sequence.
<>	A$<>B$	The strings A$ and B$ are not equivalent.

When two strings of unequal length are compared, the shorter string (of length n) is compared with the first n characters of the longer string. If this comparison is not equal, that inequality serves as the result of the original comparison. If the first n characters of the strings are the same, the longer string is greater than the shorter string.

A null string (of length zero) is less than any string of length greater than zero.

3.4 Basic file name conventions

A file is a program or a collection of data stored on a disc-type storage device. Files stay in the system permanently unless they are explicitly removed. Files are identified by a File Descriptor, hereafter referred to as 'fd' in the formats shown in this manual.

The file descriptor contains the file name.

The format can be expressed in two ways:

1. <filename>

2. <filename>.<ext>

where:

filename	Name of the file. It may be from one to twelve characters, the first alphabetic and the remaining alphanumeric.
ext	Ext is one to three alphabetic characters, describing the type of data within a file. Refer to the D-NIX Manual for details.

For Basic III commands (SAVE,UNSAVE,LIST, MERGE,LOAD, and RUN) the system recognizes two different types:

.bac BAC-BASIC compressed form
.bas - BASIC uncompressed ASCII form

When the file type is omitted in these commands (except LIST and MERGE),BASIC will look for type .bac (compressed form) first and then .bas (uncompressed form). If type .bas is specified only .bas will be searched for in the library. The SAVE command produces the type .bac as default, while the LIST and MERGE commands use .bas if the type is not explicitly given.

Examples:

Examples of legal file descriptors are:

LIST PR:	The current program is displayed on printer.
MERGE main	Merges lines from file main from the current Directory into the current program.
LOAD pack/main	Loads program main from subdirectory pack into working memory. main.bac is searched first, and if not found, main.bas is searched.
UNSAVE pack/main.bac	Deletes program main.bac from subdirectory pack.

4. CONTROL COMMANDS

Contents

4. CONTROL COMMANDS

4.1 Introduction

It is possible to communicate with the Basic III interpreter entering direct commands from the keyboard. Also, certain other statements can be directly executed when they are given without statement numbers.

Commands have the effect of causing Basic III to take immediate action. A Basic III language program, by contrast, is first entered into the memory and then executed later when the RUN command is given.

When Basic III is ready to receive a command, the promt *basic* is displayed on the screen. Commands should be typed without any line numbers.

After a command has been executed, the user will either be prompted for more information, or Basic III will again be displayed. This indicates that Basic III is ready for more input, either another command or program statements.

Example:

```
        ..
        ..
        ..
        100
        110
        <command>
        *basic*
        ..
        ..
        <command>
        ..
        ..
        *basic*
```

4.2 Control Commands

Commands control the editing and execution of programs and allow files to be manipulated. Each command is identified by a keyword at the start of the line. Keywords are shown in upper-case letters. All characters of the keyword are mandatory.

Table 4-1 lists the Basic III control commands described in this section along with a short description for each.

Table 4-1. Basic III Control Commands

Command	Description
AUTO	Generates line numbers automatically.
CLEAR	Clears all variables and closes all files.

Command	Description
CONTINUE (or CON)	After CTRL C operation this command (CONTINUE) restarts the program on the line where it stopped.
ED	Gives program editing facilities.
ERASE	Deletes blocks of lines from a Basic III program.
LIST	Outputs a program to a specified file or device.
LOAD	Loads the program requested into memory from a specified device.
MERGE	Inputs lines from a program on disc to the current program.
NEW (or SCR)	Deletes the current program and resets all modes to their default value.
RENUMBER (or REN)	Enables recording of lines.
RUN	Executes a Basic III program.
SAVE	Stores the current program on a disc.
STATUS (or STAT)	Gives information about interpreter modes and program, data sizes.
UNSAVE	Deletes a non-protected program from the disc.

The following sections describe the function, type, format, arguments and use of each of the above commands. Examples are included to show how the command can be used. Errors may occur when using a command incorrectly or syntaxically wrong. A complete list of error messages is shown in Appendix B.

4.2.1 Control Commands

AUTO

Function: Generates line numbers automatically after each carriage return.

Mode: Direct

Format:
1. AUTO
2. AUTO <line no.1>
3. AUTO <line no.1>,<step>

Arguments: Line no.1 specifies the start line and step specifies the step value.

Both 'line no.1' and 'step' are optional. If no arguments are given, then the line numbering starts with the next whole loth number (i.e., 10, 20, 30, etc.) after the existing line numbers. The step is set to 10 if the new step is not included.

Use: Auto facilitates freedom from line numbering. It is continuously available during the programming work. Automatic line generation stops when the carriage return is entered as first character on a line. If a line entered causes an error message, automatic line numbering is stopped and the line can be edited. The line numbering can be started by a new AUTO command.

The automatic line numbering can be overridden by entering a line number anyhow before the statement. This doesn't stop the AUTO line numbering mode, instead it will re-prompt with the same line number once again.

Examples: Ex. 1

AUTO 10,5

The first line number will be 10 and the line number will be incremented by 5 for each line.

```
AUTO 10,5
10 LET A=1
15 - - -
20 - - -
25 - - -
```

Ex. 2

```
AUTO
10 INPUT "CYLINDER HEIGHT =", H
20 INPUT "CYLINDER RADIUS =", R
30 PRINT "CYLINDER VOLUME -", 2*PI*R*H
40 END
50
*basic*
NEW
AUTO 50
50 INPUT "CYLINDER HEIGHT =", H
60 INPUT "CYLINDER RADIUS =", R
70 PRINT "CYLINDER VOLUME -", 2*PI*R*H
80 END
90
*basic*
NEW
AUTO 100,5
100 INPUT "CYLINDER HEIGHT =",H
105 INPUT "CYLINDER RADIUS =",R
110 PRINT "CYLINDER VOLUME -",2*PI*R*H
115 END
120
*basic*
```

CLEAR

Function: Clears all variables and closes all open files.

Mode: Direct

Format: CLEAR

Action: CLEAR does not affect the existing program which is still left in memory. CLEAR is necessary before changing the precision with SINGLE or DOUBLE if variables have already been allocated.

Example:

```
10 A%=1234%
20 END
RUN
*basic*
; A%
 1234
*basic*
CLEAR
*basic*
; A%
 0
*basic*
A%=4567%
*basic*
; A%
 4567
*basic*
```

CON

Function: Continues program execution from where it was stopped by either CTRL C entered twice or a STOP statement.

Mode: Direct

Format: CON

Action: Execution of "CON" causes the program to restart at the line at which it stopped.

Use: A variable may be displayed and changed using a direct mode statement before "CON" is used. If the program is edited or an "END" statement caused the program to be terminated, the "CON" command will cause an error and should not be used.

Example:

```
10 FOR I = 1 to 10000
20 ;1;
30 NEXT I
40 END

RUN
1 2 3 4 5 ...

CTRL C
CTRL C (enter twice)
CON
```

.... (Continue Printing)

basic

EDIT

Function:	Allows a previously entered program line to be edited.
Mode:	Direct
Format:	ED <line no.>
Argument:	Line no. is the line to be corrected. If line no. is omitted the first line in the program is edited.
Use:	Once the command is entered, the line specified will be displayed. The cursor is positioned after the last character on the line. The following terminal keys will become active and can be used at this point.
Note:	In the description of editing functions (sections 2.3, 2.4, 4.2) the default control characters are used and should be replaced by the character or function key defined in "usr/etc/bascap" or "bascap".
CTRL-H (Backspace)	Moves the cursor to the left of the current cursor position.
CTRL E (oct 005)	Erases with the cursor on the character to erase.
CTRL-L	Moves the cursor one position to the right for each touch of the key.
CTRL K (oct 013) CTRL N (oct 016)	With the uparrow key (CTRL K) and downarrow key (CTRL N) you can step up and down the program line by line in edit mode. All lines passed (and possibly edited) are entered into memory. The intension is to simplyfy editing of a program area without having to specify linenumbers for each line.
CTRL-D (oct 004)	Kills the characters from cursor to end of line.
CTRL-X (oct 030)	Kills the line in the editor buffer (not in memory if it has been entered earlier)
CTRL-I mode	Enter Insertion mode. Written characters will be placed after the the cursor until an CTRL-U (oct 025) or Return (oct 015) is given. After ESC editing can continue, but after Return the line is entered into memory.

If an error message is displayed, when entering a line, the cursor moves to the error position on the line. Edit the erroneous line with CTRL-I etc. Pressing CTRL-X at this point negates any changes made after entering the ED-command.

When giving the ED command, all files are closed and variables zeroed. Continued execution with CON can not be done.

Any character (letters, digits and other printable characters) with an octal code between 040 and 0177 replaces the current character when written.

Examples:

```
LIST
10 A$="1.4726"
20 B$="7.75"
30 ;ADD$(A$,B$,4)
*basic*
```

1. To change 1.4726 to 1.26 in line 10 do the following:
   a. ED 10
   b. Depress CTRL-H (leftarrow) four times
   c. Depress CTRL-E (Erase) twice
   d. Depress RETURN key

2. To insert 423 before 75 in line 20, do the following:
   a. ED 20
   b. Depress CTRL-H (leftarrow) two times
   c. Depress CTRL-I (enter insert mode)
   d. Enter 423
   e. Depress RETURN key

```
LIST 20
20 B$ = "7.42375"
```

ERASE

Function: Deletes blocks of lines from the current program.

Mode: Direct

Format: ERASE <argument>

Arguments: Argument can be the single line number or a range of line numbers to be listed. A single line number can have a "-" appended to or before it (e.g., 10-, -10) to designate all lines up to 10 or from 10 to the end of the program are to be listed, respectively.

Use: All lines between and including the two line numbers are removed.

Examples:

```
ERASE 20-200 ! ERASE LINES 20 UP TO AND INCLUDING
               200
ERASE -100 !   ERASE ALL LINES UP TO AND INCLUDING
               LINE 100
ERASE 50- !    ERASE FROM LINE 50 TO END OF PROGRAM
```

LIST

Function: Lists all or part of the current program to the console, printer or to a file.

Mode: Direct

Format:

```
1. LIST  [fd] [,argument]
2. LIST  [argument]
3. LIST  <PR:> [,argument]
```

Arguments: fd is the file descriptor as previously defined in Section 3.4.

Argument can be the single line number or a range of line numbers to be listed. A single line number can have a "-" appended to or before it (e.g., 10-. -10) to designate all lines up to 10 or from 10 to the end of the program are to be listed, respectively.

PR: specifies that the lines will be listed on the printer.

Use:

1. LIST pgm/xyz
Saves a program in an uncompressed way on the disc in directory 'pgm' under specified file name xyz.bas. Note that only a file saved with LIST can be accessed by a utility outside of Basic III. The MERGE command also requires a file in LIST form. While loading a file, saved with LIST, BASIC checks for syntactical errors and gives the operator a possibility to correct erroneous lines. (Compare the note under the LOAD command).

2. LIST
The entire program is listed.

3. LIST 100
Line 100 is displayed on the screen.

4. LIST 100-1000
All lines between 100 and 1000 inclusive are displayed on the screen.

5. LIST <PR:>
The entire program is output on the printer.

6. LIST -1500
All lines up to 1500 are listed.

7. LIST subfil,2000-
All lines from 2000 through the last line are listed to the file 'subfil.bas'.

Note: Large volumes are displayed on the screen one page (screen) at a time. The next line will be displayed when you press the space bar. The next page if you press CTRL-N.A long listing may be stopped by pressing CTRL-C, RETURN or entering any Basic III command/statement.

Examples:

```
LIST acct/payroll ! SAVE FILE 'payroll' in directory
'acct'
LIST ! LISTS THE ENTIRE PROGRAM ON THE SCREEN
LIST 100 ! LISTS LINE 100
LIST 100 - 500 ! LISTS LINES 100 TO 500
LIST PR: ! LISTS THE ENTIRE PROGRAM ON PRINTER
LIST PR:,100-200 ! LISTS LINES 100-200 ON PRINTER
```

LOAD

Function: Loads a Basic III program from external storage into working storage.

Mode: Direct

Format: LOAD <fd>

Arguments: fd is the file descriptor as previously defined in Section 3.4.

Note that when the file type is omitted, the computer will look for type .bac (compressed form) first and then .bas (uncompressed form).

If type .bas is specified, only .bas will be searched for in the directory.

Use: Loads the specified file after having cleared the working memory.

All open files are closed, the program area and buffers are reset. All variables are erased.

Note: If the file has been saved in un-compressed form with the LIST command, BASIC does syntactical checks of each line as if the lines were entered by the operator. Erroneous lines are reported, but are still loaded as comment lines, which may be edited by the operator. The time to load a program in compressed form is much shorter than when loading from an un-compressed file.

The code in compressed files, saved with the SAVE command, depends on the BASIC version and may not be possible to load with another BASIC version. Save the program with LIST for compatibility. The first byte in a file with compressed code contains the BASIC version number.

Examples: Ex. 1
LOAD test/abc200

Filename abc200 in the subdirectory 'test'is read, not to the END statement, but the entire file.

Ex. 2
LOAD mast
LOAD mast.bas

Filename 'mast' in un-compressed format is to be loaded into working storage.

MERGE

Function: Merges lines from a file in un-compressed form into the program.

Mode: Direct

Format: MERGE <fd>

Arguments: fd is the file descriptor as previously defined in Section 3.4.

Use: The numbered lines from the specified file are inserted in line number sequence in the current program. The lines are validated on input. New lines are inserted in line number sequence. If a new line has the same line number as an existing line then the old line is replaced by the new. All variables are initialized.

The entire file is read.

Note: The program being merged must have been saved using the LIST command.

Example: Existing program xray

```
LIST xray
5   Y=1
10  PRINT
20  FOR L=1 TO 10
30    PRINT L TAB(Y) "I";
40    READ Y
50  FOR I=1 TO Y
60    PRINT " * ";
70  NEXT I
80  PRINT
90  PRINT TAB(Y) "I"
100 NEXT L
*basic*
```

The following program file is stored on an external disk under the name 'table'

```
200 DATA 5,4,0,3,1
300 DATA 10,15,28,15,6
999 END
```

The commands: LOAD xray
MERGE table
add lines 200 to 999 into the existing program.

```
LIST xray
5  Y=1
10  PRINT
20  FOR L=1 TO 10
30    PRINT L TAB(Y) "I";
40    READ Y
50    FOR I=1 TO Y
60       PRINT " * ";
70    NEXT I
80    PRINT
90    PRINT TAB(Y) "I"
100 NEXT L
200 DATA 5,4,2,3,1
300 DATA 10,15,28,15,6
999 END
```

NEW

Function: Clears the user's program area from working storage.

Mode: Direct

Format: NEW

Use: Clears working storage and all variables and resets the pointers. The effect of this command is to erase all traces of the program from memory and to start over.

All open files are closed.

Use this command before typing in a new program.

Note: The SCR command can also be used. It works just like NEW.

Example:

```
..           Existing program
RUN

*basic*
NEW
..           Type in a new program
..
..
RUN          Run the second program
```

RENUMBER

Function: Changes the line numbering in the current program.

Mode: Direct

Format:
```
1. REN
2. REN <1st line no.><,increment>
3. REN <1st line no.><,increment><,start line
                                    -last line>
```

Arguments: 1st line no. is required in formats 2 and 3 above and is the number to be given to the first line. The default is 10.

Increment is required in formats 2 and 3. It is the increment desired between lines. The default is 10.

Start line-last line is the range of lines to be renumbered. As format 3 shows, both "1st line no." and the "increment" must be specified.

Use: All line references in the program will be changed according to the REN command.

Any references to line numbers in GOSUB, GOTO, IF, ON and RESUME statements are changed to the new numbers if necessary.

If any statement in the program references a line number and that line number does not exist, an error message is printed on the terminal. Renumbering is not done.

Example: Existing Program
```
LIST
2 A = 1
3 B = A+2
7 PRINT A,B
10 END
*basic*
REN
*basic*
LIST
10 A = 1
20 B = A+2
30 PRINT A,B
40 END
*basic*
REN 10,5
*basic*
LIST
10 A = 1
15 B = A+2
20 PRINT A,B
25 END
*basic*
REN 100,20,15-25
```

```
*basic*
LIST
10  A = 1
100 B = A+2
120 PRINT A,B
140 END
*basic*
```

RUN

Function: Loads and executes a Basic III program or executes the current program.

Mode: Direct

Format: RUN [fd]

Arguments: fd is the file descriptor as previously defined in Section 3.4.

The type specification in fd is the kind of file -
.bac
.bas

Note that when the file type is omitted, the computer will look for type .bac (compressed form) first and then .bas (uncompressed form). If type .bas is specified only .bas will be searched for in the directory.

Use:

1. RUN
   All variables and arrays in the program area are erased and all buffers are cleared. The actions of a RESTORE statement are performed and then execution of the current program is started at the lowest numbered line.

2. RUN <directory> <filename>
   The action of a LOAD command is performed. Execution of the loaded program is then started at the lowest numbered line.

Note: Compare the note under the LOAD command.

Examples: Ex. 1

```
10 READ A,B
20 LET A = A + B
30 PRINT A
40 DATA 2,3
50 END
RUN
 5
*basic*
```

If the same program is a file in the current directory,with the name 'aplusb' then:

```
Ex. 2
RUN aplusb
 5
*basic*
```

SAVE

Function: Creates a disc file and stores the current program into that file in compressed format.

Mode: Direct

Format: SAVE <fd>

Arguments: fd is the file descriptor as previously defined in Section 3.4.

Use: The command causes the program, which is currently in the working storage, to be saved in compressed form under the given file name (type .bac). No other type can be specified. The program is saved in a compressed way to enable faster loading.

Note: If the file already exists on the disk the old contents in the file will be destroyed and replaced by the new program.

If the file is saved via SAVE, the file cannot be listed by a utility outside of Basic III. If this is desired, refer to LIST command.

Compare the note under the LOAD command, concerning the difference between program files in compressed and un-compressed form..

Example:

```
10 - - -
-
-
-
-
-
-
999 END
SAVE ACT
```

SCR

Function: Clears the user's program area.

Mode: Direct

Format: SCR

Use: Clears working storage and all variables and also resets the pointers. The command erases all traces of the existing program from memory and starts over again.

All open files are closed. Use this or the NEW command before entering a new program.

Note: NEW and SCR are just different names for the same command.

Example:

```
100  ; "THIS IS A TEXT"
200  A = 4
300  ; A
RUN
THIS IS A TEST
 4
*basic*
SCR
RUN
*basic*
```

STAT or STATUS

Function: Gives information about the status of the interpreter: modes, program size, data size.

Mode: Direct

Format: STAT

Use: To check that the interpreter is set to the wanted modes.

Example:

```
STAT
NO EXTEND, FLOAT, SINGLE, LONG INT
NO PROGRAM
*basic*
10 EXTEND
RUN
*basic*
STAT
EXTEND, FLOAT, SINGLE, LONG INT
PROGRAM SIZE 35 bytes, READY TO RUN
DATA SIZE 0 BYTES
*basic*
```

UNSAVE

Function: Erases a file from a specified disk.

Type: Direct

Format: UNSAVE <fd>

Arguments: fd is the file descriptor as previously defined in Section 3.4.

Note that when the file type is omitted, the computer will look for .bac (compressed form) first and then .bas (uncompressed).

If type .bas is specified, only .bas will be searched for in the directory.

Examples:

Ex. 1
After the user has completed all work with file XYZ in the current directory, the file can be removed from storage by executing the following statement:

```
UNSAVE XYZ
```

Ex. 2
Erase file 'proga' which is a text file with type .txt.

```
UNSAVE proga.txt
```

5. STATEMENTS

Contents

5. STATEMENTS

5.1 Introduction

Statements that are used to write programmes are divided in three main groups:

1. Data control statements
2. Input/Output statements
3. Program control statements

The statements, their function, mode, format, use, and assorted examples are given in the following sections.

5.2 Data control statements

Data control statements consist of the set of statements shown in Table 5-1. Each data control statement is described in detail following this table.

Table 5-1. Data Control Statements

Statement	Description
DATA	Assigns values to variable (via READ).
DIM	Defines size of vector/matrix and strings.
DOUBLE	Designates all subsequent floating point variables and expressions to be double precision.
EXTEND	Specifies that spaces are significant, which allows for variable names of up to 32 characters in length.
FLOAT	Sets listing and input format to float mode.
INTEGER	Sets listing and input format to integer mode.
LET	Assigns a value to a variable.
LONG INT	Sets CVT% format to 32 bit integer.
NO EXTEND	Specifies that spaces are not significant and allows for variable names of one letter and an optional digit.
OPTION BASE	Defines an array's low order member position.
RANDOMIZE	Selects a random starting point for a function.
READ	Assigns value(s) to variable(s).
RESTORE	Moves data pointer.
SHORT INT	Sets CVT% format to 16 bit integer.

SINGLE	Designates all subsequent floating point variables and expressions to be single precision.

5.2.1 Data control statements

DATA

Function: Assigns values to variables; used in conjunction with READ statement.

Mode: Program

Format: DATA <value list>

Arguments: All DATA statements, no matter where they occur in a program, cause data to be combined into one data list. Commas are used as data separators while single or double quotes are used to enclose items that contains commas or spaces.

A DATA statement must be the only statement on a line.

Use: READ and DATA statements are not used without the other. See the READ statement for more information.

Examples:

Ex. 1
```
10  FOR I=1 TO 3
20  READ A$
30  PRINT"!" A$ "!"
40  NEXT I
50  END
60  DATA"HELLO: HOW ARE YOU?","TODAY IS DEC.13, 1980"
70  DATA"GOOD-BYE"

RUN

! HELLO: HOW ARE YOU?!
! TODAY IS DEC. 13, 1980!
! "GOOD-BYE"!
*basic*
```

Ex. 2
```
10  OPEN "PR:" AS FILE 1
20  READ A$
30  PRINT #1 A$
40  READ A$
50  PRINT #1 A$
60  FOR I=1 TO 6
70    READ A$
80    PRINT #1 A$
90  NEXT I
100 READ A$
110 PRINT #1 A$
120 DATA ABC,DEF,GHI,JKL,MNO,PQR,STU,WXYZ
130 DATA ABCDEFCHIJKLMNOPQRSTUVWXYZ
140 END
```

```
RUN
ABC
DEF
GHI
JKL
MNO
PQR
STU
WXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
*basic*
```

DIM

Function: Defines the maximum number of elements in a vector or in a matrix. Also defines a string's maximum length.

Mode: Direct/Program

Format:

1. `DIM <variable(n)>, [variable(n,m),]...`
2. `DIM <stringvariable(n)>, [stringvariable(n,m),]..`
3. `DIM <stringvariable(s,...)> = <expression>`
4. `DIM <variable(j:k)>`

Arguments: n and n,m are the maximum subscript values. The lower limit is either 0 or 1 depending on the most recent OPTION BASE statement. The default value is 0.

s is a one or two dimensioned subscript (more than two is possible; however, memory constraints may not permit its use).

expression is the maximum variable length.

j and k are the upper and lower subscript values. j may be negative.

The lower limit (0 or 1) indicated above can be overridden individually for each index. This is done by replacing the single maximum index for each dimension by two values (j:k) separated by a colon (format 4). Note that also negative values of 'j' may be used.

Note: A dimensioned variable can be redimensioned only if the new DIM statement defines a smaller dimension.

Use: All values used in DIM statements will be truncated to integer.

If a subscripted variable is used without appearing before in a DIM statement, it is assumed to be dimensioned to length 11 in each dimension (0-10).

The first element of every matrix is assumed to have a subscript of zero unless it is overridden by using format 4 above. All variables have a value of zero until it is assigned a value.
Vector and matrix elements can be treated as ordinary variables in the program.

A non-dimensioned string variable's max length is automatically set to the current length the first time the string is assigned a non-null value ("<>").

If less than 80 characters are used then a standard length of 80 characters is assigned.

Example:
C (1,1) = A (10,20) + B(4,7)

adds the two elements A (10,20) and B (4,7) into a new element C (1,1) in the matrix C.

The following DIM statements for strings are available:

DIM A$(N)	Defines a string vector with N + 1 strings A$(0) - A$(N). Each string has its own automatic max length. (See above.)
DIM A$(N)=I	As above but each string's max length is forced to I characters.
DIM A$(N,M)	Defines a string matrix with (N+1)*(M+1) strings each with its automatic max length.
DIM A$(N,M)=5	As the matrix above but each string's max length is forced to 5 characters.
DIM A$=I	Forces the max length of the single string A$ to I characters.
DIM B$(-N:M)=200	Defines a vector of (N+M+1) strings, where each string is 200 bytes.

The following DIM statement for arrays is available:

DIM A(-N:M)	Defines a vector with elements A(-N) to A(M) which are totally independent of the current lower limit.

Examples:

```
10   DIM  X(5), Z(4,3), A (10,10)
12   DIM  A4 (100)
14   DIM  A$(20), B$(10,20)
16   DIM  C$(40) = 4
18   DIM  D$(10,10) = 8
20   DIM  Q$ = 253%
30   DIM  A(-2:2) ! YIELDS VECTOR WITH FIVE
40   !    ELEMENTS A(-2),A(-1),A(0),A(1) and A(2)
50   DIM  B$(-3:4)=300
```

DOUBLE

Function: Sets double precision mode. Changes all variables and expression with floating point numbers to double precision (15 digits).

Mode: Direct/Program

Format: DOUBLE

Use: The DOUBLE statement should be placed before the variables are used in the program and cannot be changed when the program has been started by RUN. This change can be made when a program line has been edited or the CLEAR or NEW command has been used. The default precision is SINGLE.

Note: AS standard, BASIC is delivered with SINGLE as the default mode at start up. SINGLE and DOUBLE cannot be mixed in the same program.

Also calculations of functions like SIN(X) etc. are done with a true higher precision if DOUBLE is selected.

Example:

```
NEW
*basic*
10 DOUBLE
20 INPUT A
30 PRINT A
40 END
RUN
? 123456789
123456789
*basic*
```

EXTEND

Function: Specifies that spaces are significant and allows for extended length variable names.

Mode: Direct/Program

Format: EXTEND

Use: In the EXTEND mode, Basic III requires spaces to delimit names and functions at input of program lines, unless the adjoining character is a line number or an arithmetic operator (- + * ** /). Variable names can be any length up to 32 characters; all characters are significant. This will allow for more readable and understandable programs.

Note: The default mode is NO EXTEND.

If key words are written without spaces they may be mistaken for long variable names.

BASIC switches to EXTEND mode, at the moment when an EXTEND statement is entered as a program statement.

When a line with long variable names is listed with the LIST or ED command, the mode is automatically switched to EXTEND.

The EXTEND and NO EXTEND program statements should not both be used in the same program.

Example:

```
10 EXTEND
20 LET SUBTOTAL=UNITS*UNITPRICE
```

FLOAT

Function: Interprets all numbers without a suffix as floating point. Integers must have a "%" suffix.

Mode: Direct/Program

Format: FLOAT

Use: As standard, BASIC is delivered with FLOAT as the default it mode at start up,in which case it is not necessary to specify it before or during program entry. Variables to be interpreted as INTEGERS must be written with the "%" suffix. Suppose a program was entered and saved in the INTEGER mode. If FLOAT was entered prior to the loading of this program, all variables entered without the % suffix (e.g., A=12.456) would be interpreted as floating point variables. Refer to example below and INTEGER statement for additional information.

Note: BASIC switches to FLOAT also at the moment when a FLOAT statement is input in a program line.

Examples:

```
Ex.1
9   OPEN "PR:" AS FILE 1%
10  A=12.345
20  B=123%
30  C%=B
40  D1%=A
50  PRINT #1% A,B,C%,D1%
60  END
SAVE TEXT
RUN
12.345         123             123          12
*basic*

Ex.2
NEW
*basic*
10 REM RUN FLOAT PROGRAM AS INTEGER
```

```
20 A=10.532
30 B=145%
40 C%=B
50 D2%=A
60 ;A,B,C%,D2%
70 END
LIST procb
*basic*
NEW
*basic*
INTEGER
LOAD procb
LIST
10 REM RUN FLOAT PROGRAM AS INTEGER
20 A=10.532
30 B=145
40 C=B
50 D2=A
60 ;A,B,C,D2
70 END
RUN
11             145            145            11
```

INTEGER

Function: Controls the sign suffix for integer and float variables when entering and listing programs. Allows conversion of program from float to integer.

Mode: Direct/Program

Format: INTEGER

Use: When a program is being entered and the INTEGER statement has been given, the programmer need not type the integer suffix %. On the other hand, all floating point variables should be marked by a decimal point suffix (.). The strings should have the usual $ suffix.

A program which is stored in text format uncompressed and contains floating point variables can be run as an INTEGER program if the command INTEGER is given prior to loading the program in ASCII format. Save the program and you have converted it into an integer program.

Note: As standard, BASIC is delivered with FLOAT as the default mode at start up.

BASIC switches to INTEGER mode also at the moment when an INTEGER statement is input in a program line.

Examples:

```
Ex. 1
80  REM LISTING FORMATS OF INTEGER
90  INTEGER
100 OPEN "PR:" AS FILE 1
110 A.=10.532
120 B.=145
130 C=B.
140 D1=A.
150 PRINT #1 A.,B.,C,D1
160 END
RUN
10.532  145  145  11
*basic*

Ex. 2
LOAD test
LIST test
9   OPEN "PR:" AS FILE 1
10  A.=12.345
20  B.=123
30  C=B.
40  D1=A.
50  PRINT #1 A.,B.,C,D1
60  END
RUN
12.345         123           123          12
*basic*
```

LET

Function: Assigns a value to a variable.

Mode: Direct/Program

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

Arguments: The use of the word LET is optional. The statement does not indicate algebraic equality but performs the calculations within the expression.

Use: The LET statement can be used anywhere in a multiple statement line.

Examples:

```
Ex. 1
10  LET  A = 5.02
20  LET  X = Y7 : Z = 0
30  LET  B9 = 5 * (X/2)
40  LET  D = (3 * A) /2 * 8

Ex. 2
10  X = 36 : A = 3 + B/C : Y = X * Z
20  A$="SMITH"
30  B$="456.72"
```

LONG INT

Function: Sets the integer format used for CVT%$ and CVT% to 32 bit integers, giving 4 byte strings as result. LONG INT is default.

Mode: Direct/Program

Format: LONG INT

Use: To specify if 2 or 4 byte integers should be used when converting between integers and strings. (Compere SHORT INT.)

Example:

```
LIST
10  LONG INT
20  A$=CVT%$(4)
30  PRINT LEN(A$)
RUN
 4
*basic*
```

NO EXTEND

Function: Disables EXTEND mode and permits spaces to be not significant.

Mode: Direct/Program

Format: NO EXTEND

Use: In NO EXTEND mode, variable names can only be input as one letter and one optional digit. Spaces are not significant at input. The default mode is NO EXTEND.

BASIC switches to NO EXTEND mode also at the moment when the NO EXTEND statement is input in a program line.

When a line with long variable names is listed with LIST or ED, the mode is automatically switched to EXTEND.

The EXTEND and NO EXTEND program statements should not both be used in the same program.

The NO EXTEND mode needs to be used only for importing programs from systems that list programs without spaces between identifiers.

Example:

```
10  EXTEND
:
:
200 INPUT "NEXT NAME:" NAME$
210 INPUT "YOUR ADDRESS:" ADDRESS$
220 IF NAME$=DEFAULTNAME$ THEN 100
:
```

```
:
300 IF ADDRESS$=LOCATIONA$ THEN PRINT "MATCH FOUND";
301 NAME$,ADDRESS$
:
:
:
410 LET B = 400
420 INPUT "NAME IN DEFAULT" A$
:
:
```

OPTION BASE

Function: Defines an array's low order member position.

Mode: Direct/Program

Format: OPTION BASE <n>

Arguments: n must be 0 or 1. The default value is 0.

Use: Option base allows the user to specify the starting subscript for an array or vector. It allows for a saving in the memory space used for working storage when the zero element of an array is not used.

Example:

```
LIST
10  OPTION BASE 1
20  DIM A$(4)
30  A$(1)="JONES"
40  A$(2)="SMITH"
50  A$(3)="WILE"
60  A$(4)="MOHAN"
:
:
100 OPTION BASE 0
110 DIM B(5)
120 B(0)=1:B(1)=2:B(2)=4
130 B(3)=8:B(4)=16:B(5)=32
:
:
```

RANDOMIZE

Function: Selects a random starting value for the function RND.

Mode: Direct/Program

Format: RANDOMIZE

Use: This statement is placed before the first random number generator call (RND) in a program. When executed the RND function selects a random starting value so that if the same program is run twice, different results will be given.

Note: Randomize should only be used once in a program.

Examples:

Ex. 1

```
LIST
10  REM A TEST OF FUNCTIONALITY
15  REM WITHOUT RANDOMIZE STATEMENT
20  REM USING RND------FUNCTION
30  REM
40  INPUT 'HOW MANY NUMBERS?'X%
50  FOR I%=1% TO X%
60    PRINT 5*RND+5
70    NEXT I%
80  END
*basic*
RUN
HOW MANY NUMBERS? 4
8.31541
6.22497
8.83965
9.68893
*basic*
RUN
HOW MANY NUMBERS? 4
8.31541
6.22497
8.83965
9.68893
*basic*
```

Ex. 2

```
10  REM A TEST OF FUNCTIONALITY OF RANDOMIZE STATEMENT
20  REM USING RND------FUNCTION.
30  REM
35  RANDOMIZE
40  INPUT 'HOW MANY NUMBERS?' X%
50  FOR I%=1% TO X%
60  PRINT #1%, 5*RND+5
65  NEXT I%
70  END
RUN
HOW MANY NUMBERS? 4
5.05425
6.39596
8.10356
6.15174
*basic*
RUN
HOW MANY NUMBERS? 4
6.15174
5.85948
8.05445
5.07878
*basic*
```

READ

Function: Assigns values to variables; used in conjunction with the DATA statement.

Mode: Program

Format: READ <variable list>

Use: READ causes the variables listed to be assigned sequential values from the DATA statements. Before the program is run, Basic III creates a data block from all the DATA statements in the order they appear. Each time a READ statement is encountered in the program, the data block supplies the next value.

READ and DATA statements are used together.
If it is necessary to use the same data several times in a program, the RESTORE statement will reset the data pointer within the data block. See RESTORE statement.

The READ and DATA statements can also be used to input string variables to a program. See Ex.1. below.

Examples:

```
Ex. 1
10  READ A$,B$,C$
20  PRINT A$,B$,C$
30  DATA CHARLIE, BOB,"""STONE"""
RUN
CHARLIE
BOB
"STONE"
*basic*
```

```
Ex. 2
50 FLOAT
100 READ A,B,C,D,X1,X2
150 DATA 3,6,1.8
200 DATA 6.83E-3,-86.4,3.14
210 PRINT "A=" A,"B=" B,"C=" C
220 PRINT "D=" D,"X1=" X1 "X2=" X2
230 END
RUN
A = 3           B = 6           C = 1.8
D = .00683      X1 = -86.4      X2 = 3.14
*basic*
```

Note: If comma, quote or apostrophe is to be read into a string it must be enclosed by quotation marks.

RESTORE

Function: Resets data pointer to enable a specific data statement to be used again.

Mode: Program

Format: RESTORE [line number]

Examples:

```
Ex. 1
60  RESTORE          Sets the DATA statement pointer
                     to the first DATA statement in a
                     program.

Ex. 2
50  RESTORE  100     Sets the DATA statement pointer
                     to the first data on line 100.

Ex. 3
10  READ A$
20  PRINT A$
30  READ A$
40  PRINT A$
50  FOR I=1 TO 6
60    READ A$
70    PRINT A$
80  NEXT I
90  RESTORE 120
100 READ A$
110 PRINT A$
120 DATA ABC,DEF,GHI,JKL,MNO,PQR,STU,WXYZ
130 DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ
140 END
RUN
ABC
DEF
GHI
JKL
MNO
PQR
STU
WXYZ
ABC
*basic*
```

SHORT INT

Function: Sets the integer precision used for CVTZ$ and CVTZ to 16 bit integers.

Mode: Direct/Program

Format: SHORT INT

Use: To specify if 2 or 4 byte integers shouldbe used when converting between integers and strings. When precision is set to SHORT INT the functions CVTZ$ and CVTZ works with 2 byte strings. (Compare LONG INT.)

Note: When SHORT INT precision is set only the CVT functions are changed, all integer calculations are performed with 32 bit integers.

Example:

```
LIST
10  SHORT INT
20  A$=CVTZ$(4)
30  PRINT  LEN(A$)
RUN
 2
*basic*
```

SINGLE

Function: Changes all variables and expressions, which are floating point numbers to single precision (6 digits).

Mode: Direct/Program

Format: SINGLE

Use: The SINGLE statement must be placed before any variables that are used and cannot be changed once the program has been started by RUN. If a line is edited or the command CLEAR is given, SINGLE may be changed to DOUBLE or vice versa. The default is SINGLE.

Note: As standard, BASIC is delivered with SINGLE as the default mode at start up SINGLE and DOUBLE cannot be mixed in the same program.

Examples:

Ex. 1

```
NEW
10  INPUT A,B
20  PRINT A,B
30  END

RUN
?12345,123456789

12345         12345678
*basic*
```

Ex. 2

```
10 DOUBLE
20 INPUT A,B
30 PRINT A,B
40 END
RUN
? 12345,123456789

 12345         123456789
*basic*
```

5.3 Input/Output statements

Input/Output statements are program instructions that enable the user to create new disk files and perform writing, reading and maintenance operations with them. Table 5-2 lists the Input/Output statements discussed in this section.

Table 5-2. Input/Output Statements

Statement	Description
CLOSE	Terminates I/O between the Basic III program and a peripheral device.
DIGITS	Sets the number of digits to be printed and the number of digits in the NUM$ function.
GET	Reads a specified number of characters from a binary file or from the console into a string variable.
INPUT	Fetches data from a source that is external to a program.
INPUT LINE	Accepts a line of input from a source, external to the program.
KILL	Erases a disc file.
NAME	Renames a disc file.
OPEN	Opens a file.
OPTION EUROPE	Allows periods and commas in "PRINT USING" output to be replaced by commas and periods. respectively, or by blanks.
POSIT	Positions or reads the file pointer.
PREPARE	Allocates and opens a new file.
PRINT	Writes or lists data to a specified device.
PRINT USING	Allows for formatted printing, using the TAB function.
PUT	Writes a record to a disc file in binary format.

Note that the OUT and INP functions are available for direct control of the Basic III interfaces. See section 12.

5.3.1 Input/Output statements

CLOSE

Function: Terminates input/output between the Basic III program and peripheral device(s) and closes the file(s).

Mode: Direct/Program

Format: CLOSE [channel no.,...]

Argument: Channel no. has the same value as in the OPEN statement and indicates the internal channel number of the file to be closed.

The CLOSE statement is used to close one or more files. If no file number is given, all files will be closed.

Note: The END statement closes all open files. Ordinary output with the PRINT instruction will cause the last buffer to be output when the file is closed.

Files are also closed at an attempt to edit with ED.

Example:

```
 5  EXTEND
10  REM CREATE A FILE
20  PREPARE "mast1" AS FILE 2
30  FOR I=1 TO 5
40  READ M$,MM,DD,YY
50  PRINT #2,M$,MM,DD,YY
60  NEXT I
70  CLOSE 2
80  DATA . . .
90  DATA . . . .
:
:
```

DIGITS

Function: Sets the number of digits to be printed and the number of digits in the NUM$() function.

Mode: Direct/Program

Format: DIGITS <value>

Argument: Value is a number representing the printing accuracy.

Use: A number displayed by PRINT is rounded off to the nearest value for the last digit. Values too great to be displayed in this form are printed in exponent form with the specified number of digits.

The same number of digits will be the result when converting a number to a string variable with the NUM$() function.

Note: DIGITS does not affect the accuracy of calculations.

Example:

```
AUTO
10 INPUT A
20 ;A
30 DIGITS 2
```

```
40 ;A
50 END
60
*basic*

RUN
? 1268925
1.26893E+06
1.3E+06
*basic*
```

GET

Function: Reads one or more characters from the specified file or from the keyboard into a string variable.

Mode: Direct/Program

Format:

```
1.  GET <stringvar> [COUNT bytes]
2.  GET #<channel no.>,<stringvar> [COUNT bytes]
```

Arguments: Channel no. refers to the channel number as referred by the OPEN or PREPARE statement.

Stringvar is the destination variable for the input transfer.

Bytes are the number of characters to be read from the console (format 1) or from a file (format 2) starting from the position of the file pointer. The default is one byte.

Use: GET is used to read a specified number of bytes from either the console or a disc file. The data is placed into a string variable and can then be processed. It is important to position the file pointer to the correct position before each GET. This is done via the POSIT statement.

Note that if GET reads from the console, user input will not be echoed on the screen. Once the correct number of characters is entered, they are processed without the return key being depressed.

Note: The GET statement requires the number of bytes that is specified, independent of any record format. All types of characters are input. At End-of-file all data is read into 'stringvar', which achieves a length less than the size specified by COUNT. If the file pointer already is at the end of the file, an End-of-File-error (dec) is generated.

When GET is used from the key board CTRL-C breaks the input and the CONTINUE command continues the program with the CTRL-C (3) character as the last character in 'stringvar'.

Examples:

```
Ex. 1
10  REM **USE OF GET WITHOUT COUNT**
20  GET B$ !GET ONE BYTE FROM KEYBOARD
RUN
X (Not shown on console)
;B$
X
*basic*

Ex. 2
10  GET A$ COUNT 6
RUN
AAAAAA (not shown on console)
;A$
AAAAAA
*basic*

Ex. 3
LIST
10  OPEN "dataB" AS FILE 1
20  !   POSITION FILE POINTER TO 10TH BYTE
30  POSIT #1,9
40  GET #1,A$ COUNT 10
50  !   PRINT 10TH TO 19TH BYTES IN FILE
60  PRINT A$
70  CLOSE
*basic*
```

INPUT

Function: Requests data from a source that is external to the program.

Mode: Direct/Program

Format:
1. INPUT #<channel no.><,list>
2. INPUT <"prompt text"> <list> [;]

Arguments: Channel no. refers to the channel number as defined by the OPEN or PREPARE statement.

If the "#<channel no.>" is not included (see format 2 above), the system assumes data will come from the user's terminal. When the # channel number (not 0) is defined and points to another device, then the prompting function is excluded. The data is read from a file or device assigned to that specified channel. (See OPEN statement, Section 8.9). Data requested from a file must have been placed into the file by a prior PRINT statement (See Example 2.) and each field is limited by a line feed-character or a comma-character in the file.

List contains the names of arithmetic variables, numeric array elements, string variables or string array elements.

Prompt text is a character string delimited by quotes. When included prompting messages can be specified to query the user for required information.

If the optional ; after the list in format 2 is includes, no carriage return/line-feed is echoed after the input.(Example in section 6.5 ERRCODE).

Use: During program execution, the programmer can enter data when prompted. INPUT (format 2 above) causes the terminal to pause during execution, print the prompt text and wait for the user to enter data. If no prompt text is included a question mark is displayed on the screen.

The user then enters numeric values separated by commas. The values are entered. If insufficient data is given or too much data is entered, the system displays error message No. 148 or 150, respectively (See Appendix B) and no variables are updated. Depending upon how many values are to be accepted by the INPUT command, the programmer may include a PRINT statement that reminds the user of the kind of input required. This is conveniently done with the multiple format shown in example 2, below.

Note: If CTRL-C has stopped execution while inputting data, the '?' is output again for new input, if the CON command is given. The same hold if an error has occurred and the program returns to the INPUT statement with the RESUME statement.

Examples: Ex. 1

```
AUTO
10 INPUT A,B,C
20 ;C,A,B
30
*basic*
RUN
? 1,2,3
3                1                2
*basic*
```

Ex. 2

```
LIST
10  PREPARE "FILEA" AS FILE 1
20  INPUT A,B,C
30  ;#1, A "," B "," C
40  CLOSE #1! WRITES END OF FILE
50  OPEN "FILEA" AS FILE 1
60  INPUT #1, X,Y,Z
70  ;X,Y,Z
RUN
? 5,7,9
5                7                9
*basic*
```

```
Ex. 3
10  INPUT "YOUR NAME : ?"A$
20  INPUT "YOUR ADDRESS : ?"B$
```

Is equivalent to

```
10  PRINT "YOUR NAME : ";
20  INPUT A$
30  PRINT "YOUR ADDRESS : ";
40  INPUT B$
```

```
Ex. 4
10  DIM A$(20)
20  OPEN "mast" AS FILE 3
30  INPUT #3, A$
:
:
```

The first 20 bytes will be read from file mast and be placed in string A$.

INPUT LINE

Function: Accepts a line of input from the terminal or from another external device.

Mode: DIRECT/PROGRAM

Format:
1. INPUT LINE <string variable>
2. INPUT LINE [#channel no.,]<string variable>

Arguments: Channel no. is associated with the OPEN statement and stands for a device or file as a logical unit.

String variable is any legal string variable where the text from the keyboard or from a specified file is placed.

Use: INPUT LINE causes the program to accept a line of characters from the terminal or from the specified file.

All characters belonging to the line are read - spaces, punctuation characters, and quotes.

The line termination character is line feed, but "CR", "LF" is always inserted at the end of the string.

No text string can be written with the INPUT LINE statement. This facility is only available in the INPUT statement. The PRINT statement can be used to print out the prompt text.

Examples:

Ex. 1

```
10  ;"YOUR ADDRESS? "
20  INPUT LINE A$
30  PRINT TAB(15) A$
RUN
YOUR ADDRESS? Enhagsvaegen 9, 183 30 Taeby
              Enhagsvaegen 9, 183 30 Taeby
*basic*
```

Ex. 2

```
LIST
10  INPUT LINE B$
20  B$=LEFT$(A$,LEN(A$)-2)
*basic*
```

Line 20 removes CR and LF from string B$.

KILL

Function: Erases the file, named by the string, from the user's file area.

Mode: Program/Direct

Format: KILL <"fd">

Arguments: fd is the file descriptor as previously defined in Section 3.4.

Note: A user is not allowed to KILL a file if it is write-protected.

Example:

```
LIST
10  !
20  ! THIS IS A SIMPLE EXAMPLE OF A
30  ! BACKUP PROCEDURE USING NAME
40  ! AND KILL STATEMENTS.
50  !
55  !
60  OPEN "file1" AS FILE 1
70  PREPARE "file2" AS FILE 2
80  PRINT #2, TIME$ ! PRINT TIME FIRST ON FILE.
90  ON ERROR GOTO 140
100 INPUT #1,A$
110 PRINT #2,A$
120  GOTO 100
130
140  IF ERRCODE<>14 STOP ! STOP IF NOT EOF.
150  CLOSE
160  KILL "file1" ! DELETE OLD FILE.
170
180  ! RENAME NEW FILE TO THE OLD NAME.
190
200  NAME "file2" AS "file1"
210  CLOSE
220  END
*basic*
```

NAME

Function: Renames a file on disc.

Mode: Program/Direct.

Format: NAME "<directory>fd1" AS "fd2"

Arguments: The directory <directory> is not required if you are renaming a file in the current directory.
fd1 is a string literal specifying the name of the file descriptor you want to rename. fd1 previously defined in Section 3.4.

fd2 is the new name.

Examples: Ex. 1

```
100 NAME "tst/old" AS "net"
```

The instruction NAME - AS cannot transfer a file from one device to another.

Ex. 3

```
120 NAME "NJTT" AS "NJTT1"
```

changes name of file NJTT to NJTT1 in the current directory.

OPEN

Function: Opens an existing file for sequential or random access on a file-structured device (disc) or enables communication between Basic III programs and the D-NIX shell.

Mode: Direct/Program

Format: OPEN <string expr> AS FILE <channel no> [MODE a%+b%]

Note: If MODE is excluded, the default access mode is READ and WRITE.

The possible modes are:

```
 0 %  READ  only
 1 %  WRITE only
 2 %  READ and WRITE
```

Note that accessing a write protected file requires MODE 0 %.

Arguments: - string expr corresponds to an external file specification for the file to be opened of the following types:

- String constant - <"fd"> where fd is the file descriptor as previously defined in Section 3.4.

or
- string variable - for example, A$

or
- "colon-expression" - for example "PR:"

The channel no. after AS FILE must have an integer value, in the range 1 to 250, corresponding to the internal channel number on which the file is opened.

Use: OPEN is used to open files which already exist. When more than a few items are to be read or written, then the technique used by the READ, DATA and INPUT statements is inefficient. When a sequence of data items is to be transferred, the data can be conveniently handled as a data file through the use of a "channel".

After an OPEN statement the file pointer is positioned at the beginning of the file.

A data file and a volume (or device) has both an external name by which it is identified within the system and a BASIC channel number that references the file. The OPEN statement associates the external file specification with the internal channel number.

The channel number is referred to by use of the symbol # (number sign) and is followed by the channel number. After the OPEN statement, the file pointer is positioned at the beginning of the file.

OPEN "PIPEIN:<cmd>" AS FILE #1
The OPEN statement has been modified to allow opening of pipes. A pipe looks, to the user, like a file, but can only be opened for either read or write. This restriction has been imposed to avoid deadlock situations.

Writing and reading from a file is done by use of INPUT and PRINT statements of a special form. The PRINT and INPUT formats to be used with the OPEN statements are:

```
     line no. PRINT # <channel no.>, <list>
     line no. INPUT # <channel no.>, <list>
```

The <channel no.> is the same value as the expression in the OPEN statement <channel no.> and the <list> is a list of variable names, expressions, or constants as described in the PRINT and INPUT statement descriptions.

GET and PUT are used to read from and write to a binary file respectively with random access. POSIT is used to move the file pointer before GET or PUT.

Note: To create a file, use the PREPARE statement before OPEN is used. When data is to be read from an existing file, the file should be opened with OPEN.

Examples:

Ex. 1

```
50  OPEN "test" AS FILE 1
```

Ex. 2

```
10  OPEN "data" AS FILE 2
20  INPUT #2,A
30  INPUT #2,B
40  INPUT #2,C7$
```

The values of the variables A,B, and C7$ are read from the file, which was opened as file number 2. The values are read directly after the values last read. If reading is to be done from the beginning of the file, it must be opened again with the OPEN instruction.

Ex. 3

```
10  PREPARE "data" AS FILE 1
20  CLOSE 1
30  INPUT A,B,C$
40  ;A,B,C$
50  OPEN "data" AS FILE 2
60  PRINT #2, C$",."A",."B
70  CLOSE 2#
80  A=0: B=0: C$="
90  ;A,B,C$
100 OPEN "data" AS FILE 4
110 INPUT #4, D$,E,F
120 ;E,F,D$
130 CLOSE 4
140 END
RUN
? 12,24,HELLO
 12              24           HELLO
 0               0
 12              24           HELLO
*basic*
```

Ex. 4
To open a file for READ and WRITE

```
10  OPEN "file" AS FILE 1
20  OPEN A$ AS FILE 2
```

Ex. 5
To open a file in subdirectory 'test' for READ only

```
10  OPEN "tst/file" AS FILE 1 MODE 0%
```

Ex. 6
To open a file for WRITE only

```
10  OPEN "file" AS FILE 1 MODE 1%
```

Ex. 7

```
OPEN "PIPEIN:echo *" AS FILE #1
```

The command echo * is sent to the shell and the pipe is created. The standard output from process echo will be connected through the pipe to file #1 in the Basic III program. When the OPEN statement is executed, process echo will be started and the output can be read by the Basic III program with INPUT LINE #1 or GET #1.

Ex. 8
OPEN "PIPEOUT:print" AS FILE #2
File #2 will be connected through a pipe to the standard input of the process print. Everything written to file #2 will be sent to process print through the pipe. In this case printed on the lineprinter with format according to print.

Colon expressions

STRING REPLACEMENT in OPEN
To simplify often used OPEN strings, translation strings can be defined in the file "/usr/etc/translate.txt" or "translate.txt". This file can be created with an editor or a simple Basic III program. It consists of pairs of lines. The first contains the string to be searched for and the second the string to replace the original one with.

Example:
Instead of writing

```
   OPEN "PIPEIN:echo *.bas" AS FILE #1
```

we would like to write

```
   OPEN "LIB:" AS FILE #1
```

In file translate.txt we add the two lines

```
   LIB:
   PIPEIN:echo *.bas
```

When the OPEN statement is executed, the given string is scanned for a colon. If there is a colon (and it was not PIPEIN : or PIPEOUT:) the file translate.txt is read until a translation is found (if not, an error is indicated).

Note:
It is the string up to and including the colon which is replaced, the rest stays as it is.

The only colon expressions allowed in the replacement string is PIPEIN: and PIPEOUT:.

OPTION EUROPE

Function: Replaces periods and commas in "PRINT USING" output by commas and periods respectively.

Mode: Direct/Program.

Format: OPTION EUROPE n

Arguments: n can be either 2, 1 or 0. A value of 1 replaces periods and commas as previously specified while a value of 0 (default) negates the replacement.

The value of 2 gives the character space as separator character and period as terminal character. This is default.

Use: This statement is used before the PRINT USING statement to allow output to conform to European notation. That is, commas in numbers are replaced by periods and periods by commas.

Note: In either case, the same format control characters are used in the PRINT USING format string.

Note! that the character '~' is used as a format control character.

Example:

```
LIST
5  OPEN "PR:" AS FILE 1
10  DOUBLE
20  A=1.23456789E+06
30  ; #1 "FORMAT:"
40  A$="#######.## #,###,###.## #%###%###.##"
50  A$=A$+" #######~## #,###,###~## #%###%###~##"
60  ; #1 A$
70  FOR I=0 TO 2
80    OPTION EUROPE I
90    ; #1 USING "OPTION EUROPE = .#",I
100    ; #1 USING A$,A,A,A,A,A,A
110  NEXT I
120  END
RUN

FORMAT:
#######.## #,###,###.## #%###%###.## #######~##
 #,###,###~## #%###%###~##
OPTION EUROPE = 0
1234567.89 1,234,567.89 1 234 567.89 1234567 89
1,234,567 89 1 234 567 89
OPTION EUROPE = 1
1234567,89 1.234.567,89 1 234 567,89 1234567 89
1.234.567 89 1 234 567 89

OPTION EUROPE = 2
1234567.89 1 234 567.89 1 234 567.89 1234567 89
1 234 567 89 1 234 567 89
```

POSIT

Function: Positions the file pointer to record or byte position desired or returns the current position of the pointer.

Mode: Program/direct

Format:
1. Position file pointer statement:
   POSIT #<channel no.>,<position>
2. Read file pointer function:

   POSIT (<channel no.>)

Arguments: Channel no. corresponds to the internal channel number on which the file is opened.

Position is the number of bytes from the beginning of the file where access is to begin. Position "0" is the first byte.

The value given by POSIT() is not used until the next input or output is done.

Use: Each data file contains a pointer specifying the present position in bytes from the beginning of the file. This pointer can be read or positioned to a specific byte position using POSIT.

Format 1, above, is used to move the file pointer to a specified byte position from the beginning of the file (the first position). The first position = 0. POSIT can be used together with all file handling instructions.

Format 2, above, yields the current position of the file pointer.

If a position larger than the file size is given, an EOF error is generated at input. At output to a normal file new bytes are allocated up to the given position. Note that the error does not occur until the next input statement.

Examples: Ex. 1

```
:
80 POSIT #1,15
:
```

The file pointer is moved to position 15 (i.e. it points to the 16th character of file number 1.

Ex. 2

```
:
50 A=POSIT(1)
:
```

A=the position of the file pointer. In Example 1 above, the file pointer is in position 15, i.e. A=15.

```
Ex. 3
LIST
10  OPEN "vol1/lister" AS FILE 2
20  POSIT #2, 10
30  GET #2, A$ COUNT 5
40  PRINT A$
*basic*

Ex. 4
LIST
10  PREPARE "test" AS FILE 3
20  ;POSIT(3)
30  ;#3, "JOHN ALDER";
40  ;POSIT(3)
*basic*
RUN
0
10
*basic*
```

PREPARE

Function: Creates and opens a new file for sequential or random access on a file-structured device (diskette) with an I/O channel number internal to the Basic III program.

Mode: Direct/Program

Format: PREPARE <string expression> AS FILE <channel no.> [MODE a%+b%]

Note: MODE corresponds to the UNIX protection mask.

1%	execute	permission ,	others
2%	write	permission ,	others
4%	read	permission ,	others
8%	execute	permission ,	group
16%	write	permission ,	group
32%	read	permission ,	group
64%	execute	permission ,	owner
128%	write	permission ,	owner
256%	read	permission ,	owner

If MODE is excluded, the default access mode is READ and WRITE for owner and READ for group and others. This corresponds to MODE mask 256%+128%+32%+4%.

Arguments: The String Expression corresponds to an external file specification for the file to be opened of the following types:

- String constant - <"fd"> where fd is the file descriptor as previously defined in Section 3.4.

or

- string variable - for example, A$

The Channel no. after AS FILE must have an integer value corresponding to the internal channel number on which the field is opened. Numbers 1 through 250 are legal.

Use: PREPARE performs the same function as the OPEN statement with the exception that it does create the file if it does not exist. The use of OPEN assumes that the file exists.

The use of PREPARE on existing files destroys the contents.

Examples: Ex. 1

```
10 REM---TESTING THE USE OF PREPARE STATEMENT---
20 REM THIS PROGRAM CREATES A FILE ON THE DISC-
30 PREPARE "newfile" AS FILE 3%
40 A$="AB"
50 B$="CD"
60 C$="EF"
70 PRINT #3%, A$+B$+C$
80 POSITION #3%, 0
90 GET #3%, D$ COUNT 6
100 ;D$
RUN
ABCDEF
*basic*
```

Ex. 2
To create and open a new file with WRITE and READ permission for owner:

```
10  PREPARE "file" AS FILE 1 MODE 256%+128%
```

Ex. 3
To create and open a new:

```
10  PREPARE "file" AS FILE 1
```

PRINT

Function: Prints data to a device or a file.

Mode: Direct/Program.

Format:

1. PRINT
2. PRINT <list> [;]
3. PRINT <#channel no.> <list> [;]

Note: PRINT can be replaced in the above formats with a semicolon, e.g. ; <list>.

Arguments: Channel no. corresponds to the channel number in the statement. If omitted, the list data will be displayed on the screen.

List can contain variables, expressions or text strings. If an element in the PRINT list is not a simple variable or a constant, the expression is evaluated before the data is printed. Text strings are enclosed in quotes.

Use:

The positions on a line are numbered from 0 to the page width, which is 79 characters for the consol.

The line is subdivided into columns, fixed tabulator positions, starting in positions 0, 15, 30, 45, 60, and 75. A comma (,) after a variable or a string in the PRINT list specifies that the next element of the list will be printed in the next column. Two commas together in a PRINT statement cause a column to be skipped. If DOUBLE precision has been selected, the tabulator positions are spaced 25 columns instead of 15.

A semicolon (;) following a variable or a string in the list causes the next element in the list to be printed in the position, i.e. immediately after the previous character. If the list is terminated by a semicolon (;) no carriage return/line feed will follow the PRINT statement. In this case the next print statement continues to print to the same line.

When printing variables, one print position is reserved for the sign and an extra space in printed between two variables, separated by ; in the statement line.

A PRINT statement without any argument causes a carriage return and line feed to be printed (i.e. one blank line).

When a line is filled, the display continues on the first position of the next line.

The TAB (col) and CUR (y,x) functions are used to cause data to be printed in certain positions. These functions instruct Basic III where to print the next value of the PRINT list. If the cusor is already beyond the point given in the TAB function, it moves to the corresponding position on the next row.

If the #<channel no.> is not written, the system assumed the user's terminal. When #<channel no.> (not 0) is defined and points to another device, then the prompting function is excluded. The data is output to a file or device assigned to the specified channel. Channel 0 is directed to the consol, as if no channel number had been given.

Examples:

Ex. 1
```
110  PRINT X;Y;5
120  PRINT                    (spaces one line)
130  PRINT "VALUE= " X3, " SAM2= " A+2
```

Ex. 2

```
10  LET A = 5
20  LET B = 2
30  PRINT A,B,A+B,A*B,A-B,B-A,A/B
40  END
```

Ex. 3

```
110  PRINT TAB(2) B TAB(2*R) C
```

Ex. 4
Cursor positioning

```
PRINT CUR(5,12) "TESTSTRING";
```

Writes TESTSTRING beginning at row 5 and column 12. The ";" in the PRINT statement above specifies that no carriage return/line feed will follow.

Ex. 5

```
100  OPEN "myfile" AS FILE 2
200  PRINT #2, A","B","C;
```

Opens a disc file by the name 'myfile'. Values are written (printed) to this file.

TAB in PRINT statements.

Function:	Tabulates to the specified position on a line.
Mode:	Program/Direct.
Format:	TAB(<expression>)
Argument:	Expression is evaluated to an integer.
Note:	TAB must be preceded on the program line by a ";" or "PRINT". TAB must be used in a PRINT statement.
Use:	TAB can only be used with the PRINT statement. The first position on a line is position 1. The position specified by expression is always relative to position 1. (Compare the CUR function, where the 1:st column has number 0).

More than one TAB can appear on a line. If a comma separates each TAB, then the data being displayed will be on separate lines. If there are no commas between TABs, then the data will appear on the same line with two exceptions. If expression evaluates to a position number lower than that of the current position, that TAB will be executed at the specified position on the next line. If expression evaluates to a position greater than the page width, printing will appear on the corresponding position of the next line.

The page width is 254 characters, unless output is to the system consol, where the page width is 80 characters.

Example:

```
;TAB(1)"HHHH"
HHHH
*basic*
10  REM ***TAB SPACING***
20  F=2 : G=5

30  ; TAB(F)"X",TAB(G)"Y",TAB(G)"A",TAB(F)"B"
RUN
 X  Y
    A
 B
*basic*
```

PRINT USING

Function: Specifies the appearance (format) of printed data.

Mode: Direct/Program.

Format:

```
PRINT [#<chan.no.>,]USING<string1>[,
                    <string2>,...] <list>
```

Use: The PRINT USING statement can be used when a specific output format is desired. This situation might be encountered in such applications as printing payroll checks or accounting reports. The string may be a string variable, string expression, or a string constant, which formats the line to be printed, All the characters in the string are printed just as they appear, with the exception of the formatting characters. The list is a list of the items to be printed. The string is repeatedly scanned until the string ends and there are no values in the value list. If more than one format is included in the string, the first list item will use the first format, the second item the second format and so on. The string is constructed according to the rules listed in this section.

Note: Note that the "OPTION EUROPE 1" statement can be specified before the "PRINT USING" statement when European notation is desired. The formatting characters for this option is specified below.

If a numeric field exceeds the right margin, according to the page width, the field is printed at the beginning of the next line. The page width is 254 characters, unless the output is to the console, where the page width is 80 characters. For strings no check is done regarding the page width.

STRING FIELDS

When strings are to be printed via "PRINT USING", one of the following three formatting characters may be specified:

Character — Function

"!"

Specifies that only the first character in the given string is to be printed.

Example:
```
10  A$="LOOK"
20  PRINT USING "!" A$
RUN
L
*basic*
```

"Ön spacesÖ"

"Ö" is 'back-slash' (\) or Swedish upper case Ö with dots with the ASCII code 92 decimal.

Specifies that the first 2+n characters from the string are to be printed. If the "Ö" characters are typed without any spaces, two characters will be printed and so on.If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left justified in the field and padded with spaces to the right.

Example:
```
10  A$="LOOK" :B$="OUT"
20  PRINT USING "ÖÖ";B$
30  PRINT USING "ÖÖ ";A$,B$   (Note:   = blank space)
40  PRINT USING " Ö  Ö ";A$,B$,"!!"
RUN
OU
LO OU
 LOOK  OUT   !!
*basic*
```

"&"

Specifies a variable length string field. When the field is specified with "&", the string is output without formatting.

Example:
```
10  A$="LOOK": B$="OUT"
20  PRINT USING "!";A$;
30  PRINT USING "&";B$
RUN
LOUT
*basic*
```

NUMERIC FIELDS

The following formatting characters can be used to format a numeric field:

Character	Use
#	A number character # is used to represent each digit position. All digit positions will be filled. If the number to be printed has fewer digits than the positions specified, the number will be right-justified (preceded by spaces) in the field.

Example:
```
PRINT USING "####" 88
  88
```

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (0 if necessary). The numbers will be rounded off it necessary. Note that by including the "OPTION EUROPE 1" statement, the decimal point in a numeric field will be replaced by a comma at the print out.

Examples:
```
PRINT USING "##.##" .08
0.08

PRINT USING "###.##" 887.654
887.65

PRINT USING "##.## " 20.2,7.3,88.789,.567
20.20  7.30 88.79  0.56

DOUBLE
OPTION EUROPE 1
PRINT USING "#######.##"; 1.23456789E+06
1234567,89
```

Character	Use
+	The "+" sign may be used at either the left or the right of the numeric field. If the number is positive, the + sign is printed at the specified side of the number. If the number is negative, a - sign is printed at the specified side of the number.

Example:
```
PRINT USING "+##.## " -75.95,2.5,88.6,-.8
-75.95  +2.50 +88.60  -0.80
```

Character	Use
-	The "-" sign, when used at the right of the numeric field, prints to the right of a negative number. If the number is positive, a space is printed.If neither "+" nor "-" has been specified, the first digit position contains a minus sign for negative numbers.

Example:
```
PRINT USING "##.##- " -75.95,44.449,-8.01
75.95- 44.45   8.01-
```

**	The "**" placed at the beginning of a numeric field fills the unused spaces in the leading portion with asterisks. The "**" also specifies positions for two more digits (termed "asterisk fill").

Example:
```
PRINT USING "**#.#  " 22.39,-0.8,543.1
*22.4  *-0.8  543.1
```

$$	When the $$ is used at the beginning of a numeric field a $ sign is printed in the space immediately preceding the number printed. Note that $$ also specifies positions for two more digits, but that the $ sign itself takes up one of these spaces.

Example:
```
PRINT USING "$$##.## " "123.45"
$123.45
```

**$	The combination "**$" at the beginning of a format string combines the effects of ** and $$. Leading spaces will be filled with asterisks and a dollar character will be printed before the number. "**$" specify three more digit positions, one of which is the dollar character.

Example:
```
PRINT USING "**$##.##" 2.34
***$2.34
```

, or %	A ',' or '%' to the left of the decimal point, with at least one '#' between '.' and ',' or '%' in a formatting string causes a comma or a space to be printed to the left of every third digit to the left of the decimal point. A comma at the end of the format string is printed as part of the string. This comma serves as the delimiter between two numbers. A ',' or '%' specifies one digit position. Note that by including the statement, "OPTION EUROPE 1", the decimal point '.', in a numeric field will be replaced by a comma ',' and the ',' will be replaced by '.'.

Examples:
```
PRINT USING "###,#.##" "1234.5"
1,234.5

PRINT USING "#,###.##" "1234.5"
1,234.5,

PRINT USING "###%#.#" "1234.5"
1 234.5

DOUBLE
OPTION EUROPE 1
PRINT USING "#,###,###.##" "1234567.89"
1.234.567,89
```

```
DOUBLE
OPTION EUROPE 1
PRINT USING "#%###.#" "1234.5"
1 234,5
```

^^^^

"^" = Up-arrow, Upper case german U with the ASCII code 94 decimal.

Four up-arrows may be placed after the digit position characters to specify exponential format. The four up-arrows specify the position of E+xx. Any decimal point position may be specified; the exponent will be adjusted. Unless a leading + or leading or trailing + or - are specified, one digit position at the beginning of the number will be used to print the minus sign.

Examples:
```
PRINT USING "##.##^^^^" 123.45
 1.23E+02
PRINT USING ".####^^^^" -777777
%-777777                    <No room for minus sign>
PRINT USING "+.##^^^^" 234
+.23E+03
```

_

Underscore "_" has ASCII code 95 decimal.

An underscore in the format string enables printing characters, otherwise decoded as print control characters. The underscore causes the next character to be output as a literal character. The literal character itself may be an underscore if the format string contains a double underscore "__".

Example:
```
PRINT USING "_!##.##_!" 45.67
!45.67!
```

"~"

"~" is lower case german u (ASCII value 126 decimal)

A "~" may replace the decimal point, the ".", in a numeric field format to insert a blank where the "." (or "," with OPTION EUROPE 1) was to be in the output. This can be used, for example, when printing on special forms.

Examples:
```
DOUBLE
PRINT USING "#%###%###~##" 1.23456789E+06
1 234 567 89
DOUBLE

OPEN "PR:" AS FILE 1
PRINT #1,"INVOICE" TAB(30);"$ cents"
PRINT #1,STRING$(36,ASCII("-"))
PRINT #1,"200  Cookies" TAB(26);
PRINT #1, USING "#####~##" 21.15
```

```
INVOICE                       $ cents
-------------------------------------
200  Cookies                   21 15
```

Note: Note that if the number to be printed is larger than the specified numeric field. a percent character is printed before the number and the field is printed in standard format. A percent character is printed also if rounding causes a number to exceed the field.

```
Examples:
PRINT USING "##.##" 711.22
% 711.22

PRINT USING ".##" .999
% .999
```

EXAMPLE

The following programs illustrate the formatting rules presented in this section.

```
Ex. 1
LIST
5  OPEN "PR:" AS FILE 4 : L=4
10 DOUBLE
20 A=1.23456789E+6
30 A$='#########.## #,###,###.## #%###%###.##'
40 GOSUB 100
50 A$='#########~## #,###,###~## #%###%###~##'
60 GOSUB 100
70 END
80 !------
100 ; #L "Format:"
110 ; #L A$
120 FOR I=0 TO 1
130   OPTION EUROPE I
140   ; #L USING 'Option Europe=#',I
150   ; #L USING A$,A,A,A
160 NEXT I
170 RETURN
RUN

Format:
#########.## #,###,###.## #%###%###.##
Option Europe=0
1234567.89 1,234,567.89 1 234 567.89
Option Europe=1
1234567,89 1.234.567,89 1 234 567,89
Format:
#########~## #,###,###~## #%###%###~##
Option Europe=0
1234567 89 1,234,567 89 1 234 567 89
Option Europe=1
1234567 89 1.234.567 89 1 234 567 89
*basic*
```

```
Ex. 2
10 INPUT A$,A
20 PRINT USING A$ A
30 GOTO 10
RUN
```

(The screen displays a "?". The numeric field and value list are entered and the output is displayed.)

```
? +#.9
+9
? +#, 10
% 10
? ##,-2
-2
? +#,-2
-2
? #,-2
%-2
? +.###,.02
+.020
? ####.#,100
100.0
? ##+,2
2+
? THIS IS A NUMBER ##,2
THIS IS A NUMBER  2
? BEFORE ## AFTER,12
BEFORE 12 AFTER
? ####,44444
% 44444
? **##,1
***1
? **##,12
**12
? **##,123
*123
? **##,1234
1234
? **##,12345
% 12345
? **,1
*1
? **,22
22
? **.##,12
12.00
? **#### 1
*****1
? $####.##,12.34           (Note: not floating $)
$  12.34
? $$####.##,12.56          (Note: floating $)
$12.56
? $$.##,1.23
$1.23
? $$.##,12.34
% 12.34
? $$###,0.23
```

```
   $0
? $$####.##,0
    $0.00
? **$###.##,1.23
****$1.23
? **$.##,1.23
*$1.23
? **$###,1
****$1
? #,6.9
7
? #.#,6.99
7.0
? ##-,2
 2
? ##-,-2
 2-
? ##+,2
 2+
? ##+,-2
 2-
? ##^^^^,2
 2E+00
? ##^^^^,12
 1E+01
? #####.###^^^^,2.45678
 2456.780E-03
? #.###^^^^,123
0.123E+03
? #.##^^^^,-123
-.12E+03
? "#####,###.#",1234567.89
1,234,568.00
(Typing CTRL-C stops the program.)
```

PUT

Function: Writes a string to a file.

Mode: Direct/Program.

Format: PUT #<channel no.>,<string>

Arguments: Channel no. refers to the channel number previously defined by an OPEN or PREPARE statement.

String is either a string variable or a string expression.

Use: PUT is used to write a string variable or expression to a file. POSIT is used to position the file pointer to the desired point in the file.

Example:

```
LIST
10  PREPARE "filea" AS FILE 2%
20  ! FILEA/B SPECIFIES BINARY DATA FILE
30  INPUT "BINARY DATA?" A$
```

```
40  PUT #2%,A$
50  POSIT #2%,0
60  GET #2%,B$ COUNT 10
70  ;B$
RUN
BINARY DATA? "JOHN SMITH"
JOHN SMITH
*basic*
```

5.4 Program control statements

In previous sections, program examples have been executed top to bottom in order of their line numbers. In most applications, however, a programmer needs the flexibility of specifying alternate execution routes. For example, branching from one point of a program to another or reexecuting a given set of code for a specifying number of times may be required. Control statements are the mechanism which allows the programmer to control the flow of a program, and to interrupt and resume sequential execution at will. Some of these statements may also be used for debugging. Table 5-3 lists the control statements discussed in this section.

Table 5-3. Program Control Statements

Statement	Description
BYE	Transfers control from Basic III to the Operating System.
CHAIN	Loads and executes a program from a program currently being executed.
COMMON	Allows only integers as index in COMMON declarations.
DEF	Defines single or multi-line user defined functions.
END	Terminates executions of a Basic III program.
FNEND	Terminates a multi-statement function definition.
FOR	Provides the specifications for repetition in a program loop.
GOSUB	Directs program control to the first statement of a subroutine.
GOTO	Transfers control unconditionally to the statement with the specified line number.
IF-IFEND	Multiline IF statements, with optional else if construct.
IF...THEN...ELSE..	Executes a specified statement or transfers control to another line depending upon a stated condition.
NEXT	Denotes the end of a loop.
NO TRACE	Disables trace mode.
ON ERROR GOTO..	Specifies a user routine for error handling.

Statement	Function
ON...GOSUB...	Transfers control conditionally to one of several subroutines or to entry points to one subroutine.
ON...GOTO...	Transfers control to one of several lines depending on the value of the expression at the time the statement is executed.
ON...RESTORE	Restores the DATA pointer to one of several lines in the program.
ON...RESUME	Transfers control to one of several places in error and handling situations.
REM or !	Insert comments into a user program.
REPEAT-UNTIL	Loop with condition at the end of the loop.
RESUME	Transfers control from an ERROR subroutine.
RETURN	Transfers control in a subroutine back to the calling program or causes a return from a multi-line function.
SLEEP	Stops the running of a program for a specified number of seconds.
STOP	Stops program execution.
TRACE	Prints line numbers of designated executed program line.
WEND	Defines the limit of the WHILE loop.
WHILE	Defines the specific condition for leaving a loop.

5.4.1 Program control statements

BYE

Function:	Finishes the working session in Basic III and returns control to the operating system. If several BASIC users are in the system, the BASIC interpreter is not canceled until all users have finished or have been canceled.
Mode:	Direct/Program
Format:	BYE
Action:	BYE closes and saves any files remaining open for that user and returns control to the operating system.

Examples:

1.
```
*basic*
BYE
```

2.
```
:                           :
:       EXISTING PROGRAM    :
:                           :
:                           :
100 BYE
```

CHAIN

Function: Loads and executes a program.

Mode: Direct/Program

Format: CHAIN <"fd">

Arguments: fd is the file descriptor (a string literal) specifying the name of a disc file from which a Basic III program is to be loaded. See section 3.4 for a definition of the file descriptor.

Use: If the user program is too large to be loaded into memory and run in one operation the user can segment the program into two or more separate programs. The CHAIN instruction is used as a logical termination of one program to call the next one. Each program is called by its name. The program in the computer is erased and the new one is loaded. The lowest numbered program line is executed first as though a RUN command had been used. The CHAIN instruction is the last instruction to be executed. The last program in a chain does not need any CHAIN statement, but control is often transferred by CHAIN back to a program that allows the user to select the program to be run.

When CHAIN is executed, all open files for the current program are closed. Any files to be used in common by several programs should be opened in each program.

Note: Variables can be passed on to a CHAINed program by means of the COMMON instruction.

Example:
```
*basic*
NEW
AUTO
10 COMMON A,B$=10,C
20; "A  =" A         (Note:    = blank)
30; "B$=   " B$
40; C+A
50 END
60
*basic*
```

```
SAVE test
*basic*
NEW
*basic*
AUTO
10 COMMON A,B$=10,C
20 INPUT A
30 INPUT B$
40 INPUT C
50; A,B$,C
60 CHAIN "testa"
70
*basic*

SAVE TEST
*basic*
RUN
?1
?A
?2
 1             A            2
A=1
B$=A
3
*basic*
```

COMMON

Function: Enables variables to be passed from one program to another when the programs are CHAINed together.

Mode: Program

Format:
1. COMMON <var><,var>......
2. COMMON <var$=Length>[,var,...]

Arguments: Var is an integer or floating point variable or array.

Var$ is a string variable or array, which must have a specified length.

Use: The variables being passed must be present in the common statement and the common statement(s) must be executed before any other statement. The passed variables must be of the same type and size in all programs where the common variables are to be used.

Note: The length of common string variables must be declared. The COMMON statement replaces the DIM statement.

Common variables will only be passed to a compressed BASIC program and NOT to an un-compressed BASIC program (.bas).

While passing common variables in a CHAINing process, a check-sum test is performed on the common statements in the two programs, but the user must assure that variables are compatible in details to avoid run-time errors.

As indexes in COMMON-declarations only integers are allowed. Floating point numbers will be converted to integers when the statement is entered into memory, and will then appear as integers, for instance when listing the program.

Example: `10 COMMON A%,D(8),F$(20)=40`

See also examples in the CHAIN section.

DEF

Function: Defines single and multiple line user-defined functions.

Mode: Program

Format:

```
1. DEF FN<name> [(arguments)] = <function>
2. DEF FN<name> [<type>] [(arguments)] [LOCAL variables]
```

Arguments: name is any valid variable name.

type is optional and can be either % (or ".") or $.

arguments consist of dummy variables. They are optional. If defined, the same number of dummy variables must also appear in the FN function call.

function can be any valid arithmetic or logical expression containing numbers, variables or mathematical expressions.

LOCAL variables are temporary variables needed within a function definition. The LOCAL keyword makes possible the local variable name option. Variables should be declared local to the function in order to protect the global variables from being disturbed. This eliminates the need for using different variable names outside the function. Arrays cannot be declared LOCAL. The length of string variables specified as LOCAL, must be defined as constant in the DEF FN statement (See examples below).

The function VARPTR() can not be used on local variables or arguments within a function.

Use: Basic III allows the programmer to define user functions and call these functions in the same manner as standard functions such as SIN. User-defined functions can consist of a single line (see format 1, above) or multiple lines (see format 2). A multiple line DEF function (format 2) differs from the single line functions due to the absence of an equal sign following the function designation on the first line. Any number of arguments of any type or any mixture of types may be used including zero. Within the multiple line function definition there must be statements of the form: RETURN <expression> and FNEND.

When the RETURN statement is encountered, the expression is evaluated and used as the value of the function, and exit is performed from the definition.

The definition may contain more than one RETURN statement, as can be seen from the example 2 below.

Multiple line DEF functions can be called recursively; one multiple line function definition can refer to itself or another multiple line function definition. The same rules apply here as for the nesting of program loops. There must be no transfer from within the definition to outside its boundaries or from outside the definition into it. The line numbers used by the definition must not be referred to elsewhere in the program.

If run-time errors may occur within the function, a local ON ERROR GOTO statement may be used. At exit from the function, the error routine entry line-number will automatically be reset to the line-number, active in the calling program. If error routines are used, they must be designed to fullfil these criteria. Exit from the error routines must always be to the same function level where the error occured.

Note: Do not modify a global string variable within a function, if the string is used explicitly or implicitly in the same BASIC statement as the function. Implicit use of the string occurs when the function is used more than once in the same statement. This is because the string handling routines use pointers instead of copying the entire string each time it is referenced. Therefore the string must not be changed if it is referenced more than once. Below is an example of this, yielding an un-wanted result. To avoid this, use local strings.

```
10  DEF FNB$(Z)
20     X$=NUM$(Z)
30     RETURN X$
40  FNEND
50  X$="Number:" : A$=X$+FNB$(22)
```

A pointer to X$ is saved before the call to FNB$(). FNB$() modifies the contents of X$. A$ will be "22mber:22" instead of "Number:22"

Examples: Ex. 1
Single Line Function:
```
10  DEF FNA(X,Y)=X+X*Y
```

Ex. 2
Multiple Line Function: The function below determines the larger of two numbers and returns that number. Such use of the IF - THEN instruction is frequently found in multiple line functions:

```
10  DEF FNM(X,Y)
20    IF Y<=X THEN RETURN X
30    RETURN Y
40  FNEND
```

Ex. 3
Multiple Line Function: This example shows a recursive function that computes the N-factorial. (However, there are more efficient, non-recursive routines for the computation of N-factorial.):

```
LIST
5   EXTEND
10  DEF FNFAK(M%)
20    IF M%=1% THEN RETURN 1% ELSE RETURN M% *
 FNFAK (M%-1%)
30  FNEND
32  REM FACTORIAL FACTOR MUST BE <9
35  INPUT "VALUE FOR FACTORIAL (<9)?: ";X
40  PRINT X "-FACTORIAL EQUALS " FNFAK(X)
50  END
RUN
    VALUE FOR FACTORIAL: 4
    4-FACTORIAL EQUALS 24
*basic*
```

Ex. 4
This example shows the user of the LOCAL option.
```
LIST
10  DEF FNA(X) LOCAL A,A$=10
20  A=33: A$="LOCAL"
30  PRINT A$
40  PRINT A
50  RETURN 5*X
60  FNEND
100  A=22: A$="GLOBAL"
110  PRINT A$
120  PRINT A
130  PRINT FNA(8)
RUN
GLOBAL
  22
LOCAL
  33
  40
*basic*
```

```
Ex. 5
The next example shows a string function:
LIST
100  PRINT FNV1$("AABBCCDDEEFF",5%10%)
110  END
120  DEF FNV1$(A$,B%,C%)
130     IF B%=C% THEN RETURN LEFT$(A$,B%) ELSE
 RETURN RIGHT$(A$,C%-B%)
140 FNEND
RUN
 CCDDEEFF
*basic*
```

```
Ex. 6
   This example shows the use of the local ON ERROR GOTO
   LIST
   10 ON ERROR GOTO 100
   20 PRINT "Square root of PI * value"
   30 PRINT SQR(FNA(X))
   40 END
   50 !---------------
   100 PRINT "Only positive numbers allowed!"
   110 RESUME 20
   120 !--------------
   200 DEF FNA(X)
   210   ON ERROR GOTO 280
   220   INPUT "X= " X
   230   RETURN X * PI
   240   ! -------
   280   PRINT "Only numbers allowed!"
   290   RESUME
   300 FNEND
```

END

Function:	Terminates a Basic III program.
Mode:	Program
Format:	END
Use:	The END instruction should have the highest line number in the main program. After END there must be only subroutines and functions which exit to the main program. END closes all files.
Note:	The variables keep their values after END. END should be on a line by itself.

Examples:

```
Ex. 1
10  REM **
20  A$="5000"
30  OPEN "XRAY" AS FILE 1
40  PUT A$
50  CLOSE
60  END
```

```
Ex. 2
NEW
*basic*
AUTO
10  READ A,B,C
20  IF A=99 GOTO 60
30   ;A B C;
40   GOTO 10
50   DATA 4,5,6,1,2,3,99,99,99
60   END
70
*basic*
```

FNEND

Function: Terminates a multiple statement function definition.

Mode: Direct/Program

Note: This statement must never be reached by sequential statement execution. The function definition should be exited before this statement by a RETURN <expr>.

Example:

```
LIST
10  DEF FNMOT (X,Y)
20     IF Y >=X**3 THEN RETURN X
30     RETURN Y
40  FNEND
*basic*
```

FOR

Function: Sets up program loops by causing the execution of one or more statements for a specified number of times. NEXT statement is also necessary.

Format:

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

Mode: Program

Arguments: The variable in the FOR ... TO statement is initially set to the value of the first expression. AFTER this, the second and third expression is calculated and temporarily stored.

The statements following the FOR are then executed. The loop variable may not be a local variable in a function.

When "NEXT" is encountered, the variable is incremented by the value indicated as the STEP interval. The NEXT statement is specified separately. See NEXT.

If the variable value exceeds the value of the TO expression, the next instruction executed will be one following the NEXT statement.

If the initial value of the variable is greater than the terminal value, the loop will not be executed at all.

The expressions within the FOR statements are evaluated once upon initial entry to the loop. The test for completion of the loop is made prior to each execution of the loop.

Use: Program loops have four characteristic parts:

1. Initialization to set up the conditions which must exist for the first execution of the loop.

2. The body of the loop to perform the operation to be repeated.

3. The modification which alters some value and makes each execution of the loop different.

4. The termination condition, an exit test which, when satisfied, completes the loop. Execution continues to the program statement following the loop.

If the STEP expression is omitted from the FOR statement, +1 is the assumed value. Since +1 is a common STEP value, that position of the statement is frequently omitted.

The control variable can be modified within the loop. When control falls through the loop, the control variable retains the last value used within the loop plus the step value.

FOR loops can be nested but not overlapped. Nesting is a programming technique in which one or more loops are completely within another loop. The depth of nesting depends upon the amount of user memory space available.

The field of one loop must not cross the field of another loop.

It is possible to leave a FOR NEXT loop without the control variable reaching the termination value. A conditional or unconditional transfer can be used to exit from a loop. When reentering a loop which, was left earlier without being completed, be careful to ensure that the correct termination and STEP values are assigned.

Note: The FOR statement is especially suited for using integer variables; it results in faster loop execution.

Examples: Ex. 1
This program demonstrates a FOR - NEXT loop. The loop is executed 20 times. When the value for A is 20, control leaves the loop and displays the last value of A. A STEP value of +1 assumed since FOR contains no STEP variable.

```
10   FOR A%=1% TO 20%
20   PRINT "A=" A%
30   NEXT A%
40   PRINT "A=" A%
RUN
A=1
A=2
:
:
A=21
*basic*
```

The loop consists of lines 10, 20 and 30. The numbers A=1 to A=20 are printed when the loop is executed. After A=20, control passes to line 40 which causes A=21 to be displayed.

Ex. 2

Acceptable nesting

```
20 FOR  A = 1 TO 10
30   FOR  B = 2 TO 11
40   NEXT B
50   FOR  C = 1 TO 10
60   NEXT C
70 NEXT A
```

Unacceptable nesting

```
100 FOR A = 1 TO 10
110   FOR B = 2 TO 11
120   NEXT A
130 NEXT B
```

GOSUB

Function: Transfers control to the first of a sequence of statements that form a subroutine.

Mode: Program

Format: GOSUB <line no.>

Arguments: Line no. is the first line number of the called subroutine. Control is transferred to that line in the subroutine.

Use: A subprogram is a sequence of instructions which perform a task that may be repeated several times in a program. To call such a sequence of instructions, Basic III provides subroutines and functions.
It is a good programming rule to use functions instead of subroutines, as functions can be referenced by name and as BASIC can do a more

complete syntax check on functions than on subroutines.

A subroutine is part of a program that received control upon execution of a GOSUB statement. Upon completion of the subroutine a RETURN statement is used to exit the subroutine and continue program execution. At this point control is transferred to the statement following the GOSUB statement.

Note: The only instruction that may be used to exit a subroutine is GOSUB or RETURN. RETURN to the calling program must be with a RETURN statement to restore program pointers. If ON ERROR GOTO statements are used, they must be designed to fullfil this criteria. Exit from the error routine must be to the same subroutine level from where the error occurred.

Example:

```
:
:
150  GOSUB 1300
:
:
300  GOSUB 1300

:
:
400  GOSUB 1800
:
:
1300  REM ** SUBROUTINE #1**
1310  FOR I = J TO K
1320    LET I1 = 2 * N
1330    PRINT I1
1340  NEXT I
1350  RETURN
:
:
1800  REM ** SUBROUTINE # 2**
:
:
1900  RETURN
2900  END
```

Example with ON ERROR GOTO

```
5 DIM A$=28500
3000 ON ERROR GOTO 5000
3010 GOSUB 4000 ! Contains a local ON ERROR GOTO
3020 ON ERROR GOTO 5000 ! Required again here!
3030 X=SQR(A)
3035 PRINT "Result= " X
3040 PRINT "Loop"
3050 GOTO 3000
3500 !
4000 ON ERROR GOTO 4900 ! Local
4010 INPUT A
4020 RETURN
```

```
4800 !
4900 ! Local error routine
4910 PRINT : PRINT "Local error =" ERRCODE
4920 RESUME ! Try input again
4990 !
5000 ! Global error routine
5010 PRINT "Global error =" ERRCODE
5020 RESUME 3040
```

GOTO

Function: Transfers program execution unconditionally to a specified program line.

Mode: Direct/Program

Format: GOTO <line no.>

Arguments: Line no' is usually not the next sequential line in the program. GOTO may be written GO TO or GOTO.

Use: The GOTO statement is used when it is desired to unconditionally jump to a line other than the next sequential line in the program. It is possible to jump backward as well as forward within a program.

When written as a part of a multiple statement line, GOTO should always be the last statement on the line, since any statement following the GOTO statement on the same line will never be executed.

Note: The GOTO statement can be used in the direct mode after a pause, i.e., STOP or CTRL C. In this case, the program is continued from the statement number given.

Example:

```
:
:
110  X = 20
120  PRINT X
130  X = X + 1
140  IF X = Z THEN 900
150  GOTO 120
:
:
900  END
```

IF...IFEND

Function: Multiline IF statements, with optional else if construct.

Mode: Direct/Program

Format:

```
IF condition [THEN [:]] statement(s)
[ELIF condition [THEN] statement(s)]
[ELSE statement(s)]
IFEND
```

Arguments: Statement(s) can be a statement or several lines of statements.

In multiline IF statement explicit GOTO statements must be used if a jump to a different line is wanted.

Use: A multiline IF statement is distinguished from an ordinary IF by leaving the line empty after the condition, after THEN or by placing a colon immediately after THEN.

The test is done with respect to the condition after IF. If the condition is true, the consecutive statements are executed until an ELIF, ELSE or IFEND is reached. If the condition is false and ELIF has been used then this condition is tested and the consecutive statements executed if the condition was true.

If neither the IF condition nor any ELIF condition (if any) is found true, the optional ELSE clause is executed.

Note: In one IF-ELIF-ELSE-IFEND construction only one block of statements, at the most, will be executed. A block of statements means the statement(s) between THEN ELIF, THEN ELSE, THEN IFEND or ELSE IFEND.

A multiline IF must always be terminated by an IFEND.

The ELIF construct can be repeated as many times as wanted.

Examples:

Ex. 1

```
100  IF A<0 THEN
110  PRINT "Less than zero"
120  ELIF A==0 THEN
130  PRINT "Equal to zero"
140  ELSE
150  PRINT "Greater than zero"
160  IFEND

RUN
Equal to zero
*basic*
```

Ex. 2

```
100  INPUT "Select :" A
110  IF A=1 THEN
120      R=FNInit()
```

```
130  ELIF  A=2 THEN
140     R=FNPrint()
150  ELIF  A=3 THEN
160     R=FNEnter()
170  ELSE
180  PRINT "Wrong select"
190  IFEND
RUN
Select : 0 (R)
Wrong select
*basic*
```

IF..THEN...ELSE

Function: Transfers program control to another line or executes a specified statement depending upon a stated condition.

Mode: Program

Format: IF <condition> THEN <argument1> [ELSE argument2]

Arguments: Condition is a relational expression which may be a simple constant, variable, alphanumeric constant, string or an arithmetic expression. The test of whether or not a given condition is true is performed by means of relational operators. They permit comparisons to be performed that determine the relationship of variables, constants, or expressions to each other.

The result of the comparison is an indication of whether a given relationship between two data items is true or false, not a numerical value.

Argument1 can be a line number or a statement.
- Line number : Control is transferred to this line when the condition (relational expression) is evaluated to be true (-1).
- Statement: May be any Basic III statement(s) which is executed when the condition (relational expression) is evaluated to be true (-1).

Argument2 can be a line number or a statement. The ELSE keyword is required.
- Line number: Control is transferred to this line when the condition (relational expression) is evaluated to be false (0).

- Statement: May be any Basic III statement(s) which is executed when the condition (relational expression) is evaluated to be false (0).

Note: THEN may be replaced by GOTO in the format but the arguments are then restricted to line numbers only.

Use:

IF...THEN...ELSE is a built-in test which allows a program to determine which of two or three routes it should choose during execution.

The specified condition is tested. If the condition is met (the expression is logically true), control is transferred to the line number given after THEN or the statement given after THEN is executed. If the condition is not met (the expression is logically false), the program execution continues at the program line following the IF statement if the "ELSE" clause is not included.

THEN may be followed by either a line number or one or more Basic III statements. I Basic III statements are given and the condition is met, these statements will be executed before the program continues with the line following the IF statement. The condition applies to all statements that follow on the same line as the IF statement.

ELSE, when included, is followed either by a line number which is used as a jump address or one or more statements which are executed before the line following the IF statement. If the condition is met,. the statement between THEN and ELSE will be carried out.

When relational expressions are evaluated, the arithmetic operations take precedence in their usual order. The relational operators have equal weight and are evaluated after the arithmetic operators but before the logical operators.

The Relational Operators are:

=	Equal
<>	Not Equal
<	Less Than
>	Greater Than
<=	Less Than or Equal
>=	Greater Than or Equal

A relational expression has a value of -1 if it is evaluated to be true and zero if it is evaluated to be false. For example:

5+6*5>15*2 is true.

Relational operators can be used to perform comparisons between two strings for example, whether A$=B$.

In performing string comparisons, the system does a left-to-right comparison. This is based on tne ASCII collating sequence of the numeric codes in the characters of the strings being compared (including such characters as leading and trailing spaces).

Examples:

```
Ex. 1
170  IF A<B+3 THEN 160
180  IF A=B+3 THEN PRINT "A HAS THE VALUE " A
190  IF A>=B THEN T1=B
200  IF A$=B$ THEN PRINT "EQUAL ":A=1/B
210  IF A>B THEN PRINT "GREATER " ELSE PRINT
 "NOT GREATER"

Ex. 2
TRACE
10  REM IF...THEN...ELSE EXAMPLE
15  ;
20  INPUT "F="F
40  C=(F-32)*5/9
50  IF F>=0 AND F<=32 THEN 70
60  IF F>=212 THEN 165 ELSE 100
70  REM PATH TAKEN FOR F=0 TO 32
80  REM
90  REM
100  ; "F=" F,"C=" C ! PATH TAKEN FOR F>32 TO <212
110  GOTO 15
165  ;  REM PATH TAKEN FOR F >= 212
170  ;  "END OF TEST"
180  END
RUN

10   15
20   F=-30
40   50 60 100 F=-30  C=-34.444
110  15
20   F=21
40   50  60  70  80  90  100  F=21  C=-10.4444
110  15
20   F=38
40   50 60 100 F=38  C=655556
110  15
20   F=400
40   50 60 165
170  END OF TEST
180
*basic*
```

NEXT

Function: Terminates a program loop which began with a FOR statement.

Mode: Program

Format: NEXT <variable>

Arguments: Variable is the same variable specified in the FOR statement. Together the FOR and NEXT statements describe the boundaries of the program loop. When execution encounters the NEXT statement, the computer adds the STEP expression value to the variable and checks to see if the variable is still less than or equal to the terminal expression value. When the variable exceeds the terminal expression value, control falls through the loop to the statement following the NEXT statement.

Use: When NEXT is encountered the variable will be incremented by the internal. See FOR statement.

Example: See FOR statement.

NO TRACE

Function: Terminates the printout of line numbers initiated by TRACE statement.

Mode: Direct/Program

Format: NO TRACE

Example:

```
10   PRINT "BEGIN "
20   K=-1
30   TRACE
40   IF K>1 THEN 80
50   K=K+1
60   PRINT "NUMBER " K
70   GOTO 40
80   A=K
90   NO TRACE
100  PRINT "STOP"
RUN
BEGIN
40 50 60 NUMBER 0
70 40 50 60 NUMBER 1
70 40 50 60 NUMBER 2
70 40 80 90
STOP
```

The TRACE function is disabled before line 40 and after line 90.

ON ERROR GOTO

Function: Specifies a user routine for error handling.

Mode: Program

Format: ON ERROR GOTO <line no>

Arguments: The specified Line no. is the start of an error routine.
Compare the description in section 2.10!

Use: Normally the occurrence of an error causes termination of the user program execution and the printing of a diagnostic message.
Some applications may require the continued

execution of a user program after an error occurs. In these situations, the user can execute an ON ERROR GOTO statement within the program.

This statement is placed in the program prior to any executable statements with which the error handling routine deals.

The system will then know that a routine exists that will take over and analyze any I/O or computional error encountered in the program and possibly make an attempt to recover from that error.

The variable ERRCODE is associated with the statement and available for the user program.

For Error Codes and Messages see Appendix B.

If there are portions of the user program in which any errors detected are to be processed by the system and not by the user program, the error routine can be disabled by:

```
     line no   ON ERROR GOTO
```

without a line number following GOTO, which returns control of error handling to the system.

Note: Read the section 2.10, how to enter into and exit from an error routine.

Example:

```
10 REM THIS PROGRAM ACCEPTS ONLY POSITIVE NUMBERS.
20 ON ERROR GOTO 80
30 REM "CON:" IS OPEN AS FILE 0%
40 INPUT "POSITIVE NUMBER"A
50 Z=SQR(A)
60 PRINT "SQUARE ROOT OF:" A "IS----->" Z
70 STOP
80 FOR I=1 TO 10
85   ; CHR$(7) ! SYSTEM BEEPS
87 NEXT I
90 PRINT "ENTRY ERROR----ONLY POSITIVE NUMBERS"
95 PRINT "ALLOWED"
110 END
RUN
POSITIVE NUMBER? 25
SQUARE ROOT OF 25 IS -----> 5
STOP IN LINE 70
RUN
POSITIVE NUMBER? -10
(system beeps 10 times)
ENTRY ERROR----ONLY POSITIVE NUMBERS ALLOWED
:
:
```

ON...GOSUB...

Function:	Conditionally transfers control to one of several subroutines or to one of several entry points to one subroutine.
Mode:	Program
Format:	ON <expression> GOSUB <list of line numbers>
Arguments:	Depending on the integer value of the expression, control is transferred to the subroutine which begins at one of the line numbers listed. Execution is resumed at the line following the statement. If the value of the expression addresses a line number outside the range of the LIST, an error message will be displayed.
Use:	Since it is possible to transfer control into a subroutine at different points, the ON - GOSUB statement could be used to determine which part of the subroutine should be executed.
Note:	See also ON ... GOTO statement.

Example:

```
10     FOR X = 1.7 to 5.9 STEP .6
20     PRINT X;
40     ON X GOSUB 1300, 200, 1300, 400, 1300, 1300
50     PRINT A$
60     NEXT X
70     GOTO 9999
200    LET A$ = "SUB200"
210    RETURN
400    LET A$ = "SUB400"
410    RETURN
1300   LET A$ = "SUBI300"
1310   RETURN
9999   END
```

Control is transferred to:

```
line  200 for X = 1.7
      200       2.3
      200       2.9
     1300       3.5
      400       4.1
     1300       4.7
     1300       5.3
     1300       5.9
```

```
RUN
1.7  SUB200
2.3  SUB200
2.9  SUB1300
3.5  SUBI300
4.1  SUB1400
4.7  SUBI300
5.3  SUBI300
```

5.9 SUB1300
basic

ON...GOTO

Function: Transfers control to one of several lines depending on the value of the expression at the time the statement is executed.

Mode: Program

Format: ON <expression> GOTO <line no.1>[,line no.2,.......]

Arguments: Expression can be any legal arithmetic or logical expression.

Line no. is where control is transferred to as illustrated in the example below.

Use: ON...GOTO permits the program to respond to multiple choices. It eliminates the necessity of separate lines for each alternative. The expression is evaluated and rounded to the nearest integer. This integer is used as an index or as a pointer to one of the line numbers in the list. An error message will be generated if it is outside the range.

Example:

```
100 ON A/B GOTO 1000,1500,1700
```

transfers control to:

1. line number 1000 if .5 < = A/B < 1.5
2. line number 1500 if 1.5 < = A/B < 2.5
3. line number 1700 if 2.5 < = A/B < 3.5
4. gives error if A/B < 0.5
5. gives error if A/B > 3.5

ON...RESTORE

Function: Restores the DATA-pointer by the same selection routine as the ON-GOTO statement.

Mode: Program

Format: ON<expression>RESTORE<line no.1>[,line no.2,.......]

Arguments: Expression can by only legal arithmetic or logical expression.

Line no. is where the DATA-pointer is restored to as explained below.

Use: This statement can be used to reset the DATA-pointer to a specific point in the data buffer. The expression is evaluated and rounded to the nearest integer. This integer is used as an index to set the

DATA-pointer to the corresponding list number. An error message will be generated if it is outside the range.

Example:

```
AUTO
10  FOR X=1 TO 3
20  READ  A,B,C
30  ON X RESTORE 60,70,80
40  PRINT A,B,C
50  NEXT X
60  DATA 1,2,3
70  DATA 4,5,6
80  DATA 7,8,9
90  END
RUN
 1 2 3
 4 5 6
*basic*
```

ON...RESUME

Function: Transfers control to one of several line numbers depending on the value of the expression in error handling situations.

Mode: Program

Format: ON <expression> RESUME <line no.1>[,line no.2,...,..]

Arguments: Expression can be any legal arithmetic or logical expression.

Use: This statement is used to accomplish a conditional return from an error handling routine. The expression is evaluated and rounded to the nearest integer. This integer is used as an index or a pointer to one of the line numbers in the list. ON...RESUME is used with ON ERROR GOTO as described in Section 2.10.

Example:

```
10   ON ERROR GOTO 100
:
:
100  REM ERROR HANDLER
:
:
150  ON B RESUME 1000,2000
:
:
```

REM

Function: Inserts comments into a user's program.

Format: REM [remark]
or

! [remark]

Argument: Remark can contain any printing characters on the keyboard. The Basic III interpreter completely ignores anything on a line following the letters REM or !. No colon is needed between a statement and the remark if ! is used.

Result: Must be used as a statement.

Use: It is often desirable to insert notes and messages within a user program. Documenting a program enables easy referencing by anyone using the program. REM statements do not offset program execution.

Example: Typical REM statements are shown below:

```
10   REM ...THIS PROGRAM CALCULATES MEAN VALUES..
20   ! ***MEAN VALUES ARE AVERAGE VALUES***
30   DEF FNSEC(X)=1/SIN(X) ! DEFINE SECANT FUNCTION
```

Remarks are printed when the user program is listed.

Note: In direct mode the ! (exclamation mark) at the beginning of a line informs the Basic III interpreter to take the rest of the line as a command to a subshell.

Example:
```
!l
```
gives a listing of the files in the current directory.

REPEAT-UNTIL

Function: Initiates a loop with the termination condition at the end of the loop.

Mode: Program

Format:
```
REPEAT
statement(s)
UNTIL condition
```

Arguments: When the UNTIL statement is executed, the condition is evaluated. If the condition is false, a jump is made to the corresponding REPEAT statement, if it is true the execution continues on the next line.

Use: REPEAT should be used only in iterative loops where the loop structure modifies the values that determines the loop termination.

The REPEAT loop structure is always executed once, compare with WHILE loop structure.

Example:

```
LIST
10  REPEAT
20  INPUT  "Value >0:"  X
30  UNTIL  X>0
*basic*
RUN
Value >0:-1
Value >0: 1
*basic*
```

RESUME

Function: Transfers control to a specified line number from an error routine or to the statement which caused the error.

Mode: Program

Format: RESUME [line no.]

Arguments: Line no. specifies where program execution will continue. If it is omitted, program execution continues at the statement which caused the error.

Note: Read section 2.10 before using the statement!

If an error occurs in a subroutine or a function, it is essential that the continued execution at exit from the error routine, must be at the same subroutine or function level.

Example:

```
LIST
10  REM THIS PROGRAM WORKS FOR ONLY POSITIVE NUMBERS.
15  REM -----FUNCTIONALITY OF RESUME-----
20  ON ERROR GOTO 80
30  REM "CON:" IS ALWAYS OPEN AS FILE 0%
40  INPUT "POSITIVE NUMBER" A
50  Z=SQR(A)
60  PRINT "SQUARE ROOT OF:" A "IS-------->" Z
70  STOP
80  ; "EINSTEIN-----ONLY POSITIVE NUMBERS ALLOWED"
85  ; CHR$(7)
90  RESUME 40
100 END
*basic*
```

RETURN

Function: Transfers control back to the calling program or causes a return from a multiple line function.

Mode: Program

Format:
1. RETURN
2. RETURN <expression>

Argument: Expression is any valid Basic III expression containing constants and variables.

Use: Format 1 is used to transfer control back to the statement following the original GOSUB statement. After having reached the subroutine through a GOSUB or an ON...GOSUB statement, the subroutine is executed until the interpreter encounters a RETURN statement. Subroutines can be nested, that is one subroutine can call another subroutine or itself.

Format 2 is used when defining a multiple line DEF function. When this RETURN statement is encountered, the expression is evaluated and used as the value of the DEF function. An exit is then performed from the defintion. The DEF definition can contain more than one RETURN statement.

Examples:

```
Ex. 1
LIST
50  GOSUB 1300
:
:
:
1300  REM ** SUBROUTINE 1***
1400  LET K=1
:
:
2000  RETURN
:
:
9999  END
*basic*
```

```
Ex. 2
AUTO
10  DEF FNM (X,Y)
20    IF Y<X THEN RETURN X
30    RETURN Y
40  FNEND
50
*basic*
```

SLEEP

Function: Suspends the currently running program for a specified number of seconds. At the end of this period the program resumes execution.

Mode: Program.

Format: SLEEP <expression>

Argument: The value of the expression determines the number of seconds.

Result: Must be used as a statement.

Example:

```
10 FOR I = 0 TO 100
20 NEXT I
30 ;I
60 ;TIME$
70 SLEEP (10) !10 SECOND DELAY
80 PRINT "GOOD-BYE"
90 ;TIME$
100 END
RUN
 101       (Note:  10 second delay)
 1981-06-02 10.10.00
 GOOD-BYE
 1981-06-02 10.10.10
*basic*
```

STOP

Function: Terminates program execution.

Mode: Program

Format: STOP

Use: The STOP statement terminates the execution of the program. The variables are not reset and the open files remain open. Program execution can be continued by one of these commands: CON or GOTO.

The STOP statement differs from the END statement in that it causes Basic III to display the statement number where the program halted. It can occur several times in a single program and is recommended for debugging purposes.

Example:

```
:
:
100 STOP
```

The message displayed is:

```
stop in line 100
```

TRACE

Function: Prints the line numbers of the executed program lines.

Mode: Direct/Program

Format: TRACE [#channel no.]

Argument: Channel no. is the internal file number representing the destination where trace data is to be sent.

Use: TRACE is used when debugging a program to track the execution of the program.

Example:

```
LIST
100  OPEN "PR:" AS FILE 1%
110  A=15.345
115  TRACE #1%
120  B=153
125  IF A=0 THEN STOP
130  C%-B
135  X=A*2
140  D1%=A
145  NO TRACE
150  PRINT #1%,A B C% D1% X
160  CLOSE 1%
170  END
RUN
```

(The following text will be printed on the printer when the above program is executed:)

```
120 125 130 135 140 145
15.345 153 153 15 30.69
```

UNTIL

Function: Defines the termination condition for a REPEAT loop.

Mode: Program

Format: UNTIL <condition>

Use: See REPEAT statement in this section.

Example: See REPEAT statement in this section.

WEND

Function: Defines the limit of the WHILE loop.

Mode: Program

Format: WEND

Use: When the WEND statement is executed, control is transferred to the last non-terminated WHILE statement.

Example: See WHILE statement in this section.

WHILE

Function: Defines a specific condition for leaving a loop.

Mode: Program

Format: WHILE <relational expression>

Argument: Relational expression is some test condition.

Use: WHILE should be used only in iterative loops where the logical loop structure modifies the values that determine the loop termination. This is a significant departure from FOR loops in which control is automatically iterated.

There are many situations in which the final value of the loop variable is unknown in advance. What is desired is to execute the loop as many times as necessary to satisfy some special conditions specified by WHILE.

Examples:

```
10  WHILE X < 10
20     X = X*X + 1
30  WEND
```

Before the loop is executed and at each loop iteration the condition X < 10 is tested. The iteration continues if the result is true.

The above example is equivalent to:

```
10  IF X > = 10 THEN 40
20  X = X*X + 1
30  GOTO 10
:
:
```

6. FUNCTIONS

Contents

6. FUNCTIONS

6.1 Introduction

Functions in the context of Basic III are independent programs stored in the interpreter which perform specific mathematical, string, or miscellaneous operations. A user program can include a call to a Basic III function program whenever it requires the execution of any of these operations. These functions can save a great deal of coding time. They enable the user to include the function without having to know the details behind them.

This section discusses four types of functions:

1.	Mathematical functions	Section 6.2
2.	String functions	Section 6.3
3.	CVT conversion functions	Section 6.4
4.	Miscellaneous functions	Section 6.5

6.2 Mathematical functions

When programming, the user may encounter many cases where relatively common mathematical operations are performed. The results of these common operations are likely to be found in mathematical tables; i.e., sine, cosine, square root, log, etc. Since the computer can perform this type of operation with speed and accuracy, these operations are built into Basic III. Internal functions can be called whenever such a value is needed. For example:

```
SIN (23.*P1/180.)
LOG (144.)
```

The various mathematical functions available are listed in Table 6-1.

Note that all functions are calculated with a true higher precision if the DOUBLE precision mode has been selected.

Table 6-1. Mathematical Functions

Function	Description
ABS(x)	Returns absolute value of x.
ATN(x)	Returns arctangent of x in radians.
COS(x)	Returns cosine of x in radians.
EXP(x)	Returns exponential function (i.e. ex)
FIX(x)	Returns the truncated value of x.
HEX$(x)	Returns the hexadecimal string representation of a decimal number.

Function	Description
INT(x)	Returns the greatest integer that is less than or equal to x.
LOG(x)	Returns the natural logarithm of x (i.e., log ex).
LOG10(x)	Returns the common logarithm (base 10) of x.
MOD(x,y)	Returns the remainder of the integer division X,Y.
OCT$(x)	Returns the octal string representation of a decimal number.
PI	Returns a constant value of 3.1415927.
RND	Returns a random number between 0 and 0.999999.
SGN(x)	Returns the sign of x.
SIN(x)	Returns the sine of x.
SQR(x)	Returns the square root of x.
TAN(x)	Returns the tangent of x.
SWAP%(n%)	Returns an integer with the first and the second byte transposed.
SWAP2%(N%)	Returns an integer with the first 16 bits and the next 16 bits transposed.

Each function listed in Table 6-1 is described in detail in subsequent paragraphs.

Order of Execution

A mathematical function is executed in the following manner:

1. The operation or operations within the argument are performed.
2. The function itself is evaluated.
3. The remaining arithmetic operations in the statement are performed in their normal order or precedence.

6.2.1 Mathematical functions

ABS

Function:	Returns the absolute value of x
Mode:	Direct/Program
Format:	ABS(x)
Argument:	x is numerical
Result:	Numeric

Example:

```
; ABS(- 123)
123
*basic*
```

ATN

Function: Returns the arctangent of x'

Mode: Direct/Program

Format: ATN(x)

Argument: x is in radians

Result: Numeric

Example:

```
; ATN(5)
1.3734
*basic*
```

COS

Function: Returns the cosine of x.

Mode: Direct/Program

Format: COS(x)

Argument: x is in radians

Result: Numeric

Example:

```
10 A = .57
20 B = COS (A)
30 PRINT B
40 END

RUN
.841901
*basic*
```

EXP

Function: Returns the value of e_x where e = 2.71828 (single precision).

Mode: Direct/Program

Format: EXP(x)

Argument: $-88 < x > 88$

Result: Numeric

Example:

```
PRINT EXP (1)
2.71828
*basic*
```

FIX

Function: Returns the truncated value of x.

Mode: Direct/Program

Format: FIX(x)

Argument: x is numeric

Result: Numeric

Example:

```
PRINT FIX (-123.96)
-123
*basic*
```

HEX$

Function: Converts a decimal number into a hexadecimal string.

Mode: Direct/Program

Format: HEX$(x)

Argument: x is decimal number

Result: String

Example:

```
10 Y$=HEX$(255)
20 ; Y$
RUN
FF
*basic*
```

INT

Function: Returns the greatest integer which is less than or equal to X.

Mode: Direct/Program

Format: INT(x)

Argument: x is numeric

Result: Numeric

Use: The integer function returns the value of the greatest integer not greater than x.

INT can also be used to round to any given decimal place, by asking for:

```
     INT(X*10.**D%+.5)/10.**D%
```

Where D% is the number of decimal places desired.

If the number is negative, INT will return the largest integer less than the argument.

Examples:

Ex. 1
```
10 Y=INT(34.67)
```
The result is Y=34

Ex. 2
```
10 Y=INT(34.67+.5)
```
The result is Y=35

Ex. 3
```
10 Y=INT(-23.15)
```
The result is Y=-24

Ex. 4
```
1200  INPUT "NUMBER TO BE PROCESSED BY INT", A
1210  INPUT "NUMBER OF DEC. PLACES FOR ROUNDING", D
1220  PRINT "TRUNCATED INTEGER=" INT(A)
1230  PRINT "ROUNDED INTEGER=" INT(A+.5)
1240  PRINT "ROUNDED TO " D "PLACES="
1250  PRINT INT(A*10**D+.5)/(10**D)
1300  PRINT
1310  PRINT"ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP"
1320  INPUT A
1330  IF A < > 0 THEN GO TO 1210
9999  END
```

```
RUN
NUMBER TO BE PROCESSED BY INT? 13.56
NUMBER OF DEC. PLACES FOR ROUNDING? 1
TRUNCATED INTEGER=13
ROUNDED INTEGER=14
ROUNDED TO 1 PLACES=13.6

ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP
? 123.4567
NUMBER OF DECIMAL PLACES FOR ROUNDING? 2
TRUNCATED INTEGER=123
ROUNDED INTEGER=123
ROUNDED TO 2 PLACES=123.46

ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP
? 0
*basic*
```

LOG

Function: Returns the natural logarithm of X, ${}_e\log X$.

Mode: Direct/Program

Format: LOG(x)

Argument: x > zero

Result: Numeric

Example:
```
PRINT LOG(2)
0.693147
*basic*
```

LOG10

Function: Returns the common logarithm of x, ${}_{10}\log\ x$.

Mode: Direct/Program

Format: LOG10(x)

Argument: x > zero

Result: Numeric

Example:
```
10 A = LOG10(5)
20 PRINT 2*A
30 END
RUN
1.39794
*basic*
```

MOD

Function: Returns the remainder of an integer division of the arguments.

Mode: Direct/Program

Format: MOD(x,y)

Argument: x and y are numeric

Result: Numeric

Example:
```
; MOD(22,4)
2
*basic*
```

OCT$

Function: Converts a decimal number into an octal string.

Mode: Direct/Program

Format: OCT$(x)

Argument: x is a decimal number

Result: String

Example:

```
; OCT$(59)
73
*basic*
Y$=OCT$(59)
;Y$
73
*basic*
```

PI

Function: Returns a constant value of 3.14159 (single precision) or 3.14159265358979 (double precision).

Mode: Direct/Program

Format: PI

Result: Numeric

Example:

```
10 INPUT R
20 C = 2*PI*R
30 PRINT C
40 END
RUN
? 11
69.115
*basic*
```

RND

Function: Returns a random number between 0 and 0.999999.

Mode: Direct/Program

Format: RND

Use: RND is used to return a random number between 0 and 0.999999. The function will generate the same random number sequence every time the program is run unless a RANDOMIZE statement is placed before RND in the program.

Result: Numeric

Example:

Ex. 1
```
10 Y=RND
```

Ex. 2
```
10 Y=(D-A)*RND+A
```
Y will be assigned a random number between A and D.

SGN

Function:	Returns the sign function of X, a value of 1 preceded by the sign of X.
Mode:	Direct/Program
Format:	SGN(x)
Argument:	x is numeric.
Result:	Numeric
Use:	The sign function returns a value of +1 if X is a positive value, 0 if X is 0, and -1 if X is negative. For example: SGN(3.42) = 1,SGN(-42) = -1, and SGN(23-23) = 0.

Example:

```
1000   REM - SGN FUNCTION DEMO
1010   READ A, B
1100   PRINT "A=";A, "B=";B
1110   PRINT "SGN(A)"; SGN(A), "SGN(B)=" SGN(B)
1120   PRINT "SGN(INT(A))=";SGN(INT(A))
1200   DATA -5.43, 0.21
9999   END

RUN
A=-5.43          B=.21
SGN(A)=-1        SGN(B)=1
SGN(INT(A))=-1
*basic*
```

SIN

Function	Returns the sine of x.
Mode:	Direct/Program
Format:	SIN(x)
Argument:	x is in radians.
Result:	Numeric

Example:

```
PRINT SIN(.57)
 .539632
*basic*

PRINT SIN (PI/2)
 1
*basic*
```

SQR

Function:	Returns the square root of x.

Mode:	Direct/Program
Format:	SQR(x)
Argument:	x > zero
Result:	Numeric

Example:
```
;SQR(9)
3
*basic*
```

SWAP%

Function:	Returns an integer with the first and second bytes transposed. The bits 0-7 change place with the bits 8-15 in the integer.
Mode:	Direct/Program.
Format:	SWAP%(n%)
Arguments:	n% is an integer.
Result:	Integer

Example:
```
10 A%=512% ! ASSIGN AN INTEGER A VALUE.
20 !
30 ! THE CORRESPONDING BIT CONFIGURATION OF A% IS
35 ! 00000010 00000000 (binary) = 512 (decimal)
40 ! CONTAINING 16 BITS OR SAME AS 2 BYTES
45 ! SWAP% FUNCTION SWAPS
50 ! THESE TWO BYTES SO RESULT WILL BE
55 ! 00000000 00000010 (binary) = 2 (decimal)
60 !
70 B%=SWAP%(A%)
80 ; B%
90 END
RUN
 2
*basic*
```

SWAP2%

Function:	Returns an integer with the first and second word (16 bits) transposed. The bits 0-15 change place with the bits 16-31 in the integer.
Mode:	Direct/Program
Format:	.SWAP2%(n%)
Arguments:	n% is an integer
Result:	.Integer

Example:

```
10  A% = 1%
20  B% = SWAP2%(A%)
30  ; B%
40  END
RUN
65536
*basic*
```

TAN

Function: Returns the tangent of x.

Mode: Direct/Program

Format: TAN(x)

Argument: x is in radians.

Result: Numeric

Example:

```
10 INPUT A
20 PRINT "SIN(A)/COS(A)=" SIN(A)/COS(A)
30 ;"TAN(A)=" TAN(A)
40 END
RUN
?0.57
SIN(A)/COS(A)= .640969
TAN(A)= .640969
*basic*
```

6.3 String functions

Besides intrinsic mathematical functions (e.g., SIN, LOG), various functions for use with character strings are provided. These functions allow the program to perform arithmetic operations with numeric strings, concatenate two strings, access a part of a string, determine the number of characters in a string, and perform other useful operations. These functions are particularly useful when dealing with whole lines of alphanumeric information input by an INPUT LINE statement. The various string functions available are summarized in Table 6-2.

Table 6-2. String Functions

Function	Description
ADD$	Returns the result of adding two numeric strings.
ASCII or ASC	Returns the ASCII decimal value for the first character in a string.
A$+B$	Returns the concatenation of two strings.
CHR$	Returns a character-string having the ASCII value of arguments.

CLS	Returns a string, which (when printed) clears the screen of the terminal.
COMP%	Returns a truth value based on result of numeric comparison.
DIV$	Returns a quotient.
INSTR	Searches for and returns the Location of a substring within a string.
LEFT$	Returns left substring of an existing string.
LEN	Returns the length of a string.
MID$	Returns a substring of a string.
MUL$	Returns the result of multiplying two numeric strings.
NUM$	Returns a string of numeric characters.
RIGHT$	Returns a right substring of a string.
SPACE$	Indicates a string of spaces.
STRING$	Creates and returns a string of ASCII characters
SUB$	Returns the result of subtracting two numeric strings.
VAL	Returns the numeric value of the string of numeric characters.

Each string function is described in detail in subsequent paragraphs.

6.3.1 String functions

ADD$

Function:	Adds the values of two numeric strings to a specified number of decimal places.
Mode:	Direct/Program
Format:	ADD$(A$,B$,p%)
Argument:	A$ and B$ are numeric strings. p% when positive specifies the number of decimals in the result and when negative specifies the number of places of precision desired.
Result:	String
Note:	ASCII arithmetic calculations can operate on up to 126 characters including decimal point and sign.

Example:
```
S$="12349.178"
*basic*

PRINT ADD$(S$,"89.454",3)
12438.632
*basic*
```

ASCII

Function: Returns an integer equal to the ASCII value of the first character of a string.

Mode: Program/direct

Format: ASCII(string) or
ASC(string)

Argument: String can be a string constant or variable.

Result: Numeric

Example:
```
; ASCII("T")
 84
*basic*

10 A$="XAB"
20 ;ASCII(A$)
RUN
 88
*basic*
```

Note: The returned value is zero if A$ is empty.

CHR$

Function: Returns a character-string corresponding to the ASCII value of the arguments.

Mode: Direct/Program

Format: CHR$(n1[,n2,n3,...])

Argument: n is the ASCII decimal of the character desired.

Result: String

Example:
```
A$=CHR$(65,66,67)
*basic*
;A$
ABC
*basic*
```

CLS

Function: Returns a clear screen string. The characters in the string and the length is dependent on the terminal type used at the moment. The string is the "cl" capability read from the terminal capability database "termcap" (Refer to D-NIX manual termcap(5)).

Mode: Direct/Program

Format: CLS

Result: String

Note: The system must know what terminal type is beeing used. This is done by setting the shell variable TERM to the terminal type used and exporting it before starting the basic.

Example:
In operating system mode type:

```
$ TERM=ut100
$ export  TERM
```

Example:

```
10 PRINT CLS ;
RUN
*basic*
```

(The promt will appear in upper left corner of the screen, with the rest of the screen cleared.)

COMP%

Function: Returns a truth value based on the result of a numeric comparison of two numeric strings.

Mode: Direct/Program

Format: COMP%(A$,B$)

Argument: A$ and B$ are numeric strings.

Result: Numeric

Use: The truth values are as follows:

```
     -1 IF A$ < B$
      0 IF A$ = B$
      1 IF A$ > B$
```

Example:

```
A$="12345.6789":B$="9876.54321"
*basic*
T%=COMP%(A$,B$)
*basic*
PRINT T%
 1
*basic*
PRINT COMP%(B$,A$)
-1
*basic*
```

DIV$

Function:	Returns a quotient, A$ divided by B$.
Mode:	Direct/Program
Format:	DIV$(A$,B$,p%)
Argument:	A$ and B$ are numeric strings. A$ is the numerator and B$ is the denominator. p% when positive is the number of decimal places in the quotient and when negative specifies the number of digits of precision desired.
Result:	String
Note:	ASCII arithmetic calculations can operate on up to 126 characters, including the decimal point and sign..

Example:

```
10 C$="3.5"
20 V9$=DIV$(C$,"1.7777",3%)
30 PRINT V9$
40 END
RUN
1.969
*basic*
```

INSTR

Function:	Searches for and returns the location of a substring within a string.
Mode:	Direct/Program
Format:	INSTR(n%,A$,B$)
Argument:	A$ is a string. B$ is the substring within A$, you want to locate. n% is the character position within A$ where the search will begin.
Result:	Numeric
Use:	A value of 0 is returned if B$ is not in A$ or the character position of B$, if B$ is found to be in A$ (character position is measured from the start of the string with the first character counted as character 1).

Example:

```
A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
*basic*
PRINT INSTR(5%,A$,"OP")
 15
*basic*
```

LEFT$

Function:	Returns a substring of an existing string.
Mode:	Direct/Program
Format:	LEFT$(A$,n)
Argument:	A$ is a string. n is the character position in A$ where the substring will end. N = 0 is permitted. N must be <= the length of A$.
Result:	String
Use:	The substring will begin with the first character in A$ and end with the n:th character.

Example:

```
10  A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
20  ;LEFT$(A$,6)
30  END

RUN
ABCDEF
*basic*
```

LEN

Function:	Returns the length of a string.
Mode:	Direct/Program
Format:	LEN(A$)
Argument:	A$ is a string.
Result:	Numeric

Example:

```
PRINT LEN ("JOHN SMITH")
 10
*basic*
```

MID$

Function:	Returns a substring of a string. The function can also be used on the left-hand side of a LET statement to store new characters in the specified position in the string.
Mode:	Direct/Program
Format:	MID$(A$,n1,n2)
Argument:	A$ is a string. n1 is the character position in A$ where the substring begins. n2 is the number of characters in the substring.

n2 = 0 is permitted.
n1 + n2 must not exceed one more than the string length.

Result: String

Note: This function can also be used on the left-hand side of a LET statement. The length of the string on the right hand side must be of length n2.

Use: The characters between and including n1 through n1+n2-1 characters of A$ comprise the substring.

Example:

```
10 ;"NAME,ADDRESS? ";
20 INPUTLINE A$
30 PRINT
40 Z=INSTR(1,A$,",")
50 Y=LEN(A$)
60 ;"NAME= '" LEFT$(A$,Z-1)
70 ;"ADDRESS= " MID$(A$,Z+1,Y-(Z+1))
80 B$= "DATABOARD          FOR BUSINESS"
90 MID$ (B$,11,7)="SYSTEMS"
100 :B$
110 END
RUN
NAME,ADDRESS? DATABOARD,USA
NAME= DATABOARD
ADDRESS= USA

DATABOARD SYSTEMS FOR BUSINESS
*basic*
```

MUL$

Function: Returns the result of multiplying two numeric strings.

Mode: Direct/Program

Format: MUL$(A$,B$,p%)

Argument: A$ and B$ are numeric strings.
p% when positive specifies the number of decimal places required and when negative, the number of digits of precision desired.

Result: String

Example:

```
10 INPUT A$,B$
20 ;MUL$(A$,B$,6)
30 END
RUN

?12345.6789,987.54321
12191891.370535
*basic*
```

NUM$

Function:	Returns a string of numeric characters representing the value of n as it would be displayed by a PRINT statement.
Mode:	Direct/Program
Format:	NUM$(n)
Argument:	n is a floating point or integer variable or value.
Result:	String. The length of the string depends on the selected SINGLE or DOUBLE mode or if 'n' is an integer. Using the DIGITS statement effects the NUM$ function in the same way as the PRINT statement.

Example:

```
; NUM$(123456789012)
1.234568E+11
*basic*
```

Note: Returned string will not have any leading blanks.

RIGHT$

Function	Returns a particular substring of a string.
Mode:	Direct/Program
Format:	RIGHT$(A$,n)
Argument:	A$ is a string. n is the character position in A$ where the substring will begin. n can equal LEN(A$) + 1 which results in an empty string, but cannot be greater than A$ + 1.
Result:	String
Use:	RIGHT$ returns the characters from the n:th character through the last character in A$.

Example:

```
10 ;"NAME,ADDRESS? ";
20 INPUT LINE A$
30 PRINT
40 Z=INSTR(1,A$,",")
50 ;"NAME= " LEFT$(A$,Z-1)
60 ;"ADDRESS= " RIGHT$(A$,Z+1)
70 END
RUN
NAME,ADDRESS? DATABOARD, SWEDEN
NAME= DATABOARD
ADDRESS= SWEDEN

*basic*
```

SPACE$

Function: Inserts a string of spaces into a character string or yields a string of a specified number of spaces.

Mode: Direct/Program

Format: SPACE$(N%)

Argument: N% is the number of spaces. An error is generated if the value of N% is negative.

Result: String

Example:
```
PRINT "ABC" SPACE$(10) "DEF"
ABC          DEF
*basic*
```

STRING$

Function: Returns a string of ASCII characters.

Mode: Direct/Program

Format: STRING$(n1,n2)

Argument: n1 is the length of the string in characters.
n2 is the ASCII decimal value of the character.

Result: String

Example: Print a string of 15 *'s.

```
10 F$=STRING$(15,42)
20 PRINT F$
30 END
RUN
***************
*basic*
```

SUB$

Function: Subtracts two numeric strings and gives the result with a specified number of decimals.

Mode: Direct/Program

Format: SUB$(A$,B$,p%)

Argument: A$ and B$ are numeric strings.
p% specifies, when positive, the number of decimal places in the result and when negative, the number of digits of precision desired.

Result: String

Note:	ASCII arithmetic calculations can operate on up to 126 characters, including the decimal point and sign.

Example:

```
10 B$="9876.54321"
20 ;SUB$(B$,"98.76",5)
30 END
RUN
9777.78321
*basic*
```

VAL

Function:	Computes and returns the numeric value of a string of numeric characters.
Mode:	Direct/Program
Format:	VAL(<string>)
Argument:	String is a numeric string. The result is a floating point number. If the string contains non-numeric characters other than +, -, or ., an error routine is called.
Result:	Numeric

Example:

```
10  A=VAL("14.3E-5")
20  PRINT A
30  END
RUN
 .000143
*basic*
```

6.4 CVT Conversion Functions

CVT Conversion Functions are provided to permit floating-point and integer values to be represented in binary in files. These functions are summarized in table 6-3.

Table 6-3. CVT Conversion Functions

Function	Form	Description
CVT%$(I%)	A$=CVT%$(I%)	Maps an integer into a two or four character string (depending upon whether short integer or long integer precision is used).
CVT$%(A$)	I%=CVT$%(A$)	Maps the first two or four characters of a string into an integer. The string must have the right number of characters.

CVTF$(X)	A$=CVTF$(X)	Maps a floating-point number into a four- or eight-character string (depending upon whether Single or Double precision is used).
CVT$F(A$)	X=CVT$F(A$)	Maps the first four or eight characters (depending upon whether Single or Double precision is used) of a string into a floating-point number. The string must have enough characters; otherwise, wrong results will be returned.

Note: CVT%$ and CVT$% words on 16 or 32 bit integers depending upon the statements Long int (32 bits) and Short int (16 bits).

The above functions do not affect the value of the data, but rather its storage format. Each character in a string requires one byte of storage (8 bits); hence, characters may assume (decimal) values from 0 through 255 and no others. A 16-bit quantity can be defined as either an integer or a two-character string; two-word floating point numbers can equally be defined as four-character strings.

The four CVT Conversion Functions are described in detail in subsequent paragraphs.

6.4.1 CVT conversion functions

CVT%$

Function: Returns a two or four character string representation of an integer depending on whether Short int or Long int precision was in effect.

Mode: Program/direct

Format: CVT%$(<variable>)

Arguments: Variable can be an integer constant, integer variable or a subscripted variable.

Use: This function permits dense packing of data in records. For example, any integer value between -32768 and 32767 can be packed in a record in two characters. This would only be true for integers between -9 and 99 if the data was stored as ASCII characters.

Example:

```
LIST
 5  SHORT INT  ! CONVERT 2-BYTE INTEGERS
10  RANDOMIZE
20  DIM A$(100%)
30  ! GENERATE 10 FIVE-DIGIT RANDOM INTEGER NUMBERS
40  !
41  ; "*** INTEGERS GENERATED ***"
```

```
42  !
50  FOR I%=1% to 10%
70    A%(I%)=INT(RND*32767%)
75    ; A%(I%)
80  NEXT I%
90  !
100  ! THE INTEGERS ABOVE CAN BE.STORED INTO A FILE IN
105  ! TWO WAYS
110  !
120  ! 1. USING THE PUT AND NUM$ STATEMENTS
130  ! ...THE SIZE OF filea WILL BE 50 BYTES...
131  !
140  PREPARE "filea" AS FILE 1
150  FOR I%=1% TO 10%
157    S$=NUM$(A%(I%))
158    S$=S$+SPACE$(5%-LEN(S$))
160    PUT #1,S$
170  NEXT I%
180  CLOSE 1
185  !
190  ! 2. USING THE PUT AND CVT%$ STATEMENTS
200  ! ...THE SIZE OF fileb WILL BE 2 x 10 = 20 BYTES
201  ! ...fileb WILL BE PACKED FROM 50 BYTES TO 20
202  ! ...BYTES BY USING CVT%$.
203  !
210  PREPARE "FILEB" AS FILE 2
220  FOR I%=1% TO 10%
230    PUT #2 CVT%$(A%(I%))
240  NEXT I%
250  CLOSE 2
260  !
270  END
*basic*
```

CVT$%

Function:	Returns the integer representation of the first two or four characters of a binary string depending on whether Short int or Long int precision was in effect.
Mode:	Program/direct
Format:	CVT$%(<string>)
Arguments:	String is any string variable or constant having two characters (Short int precision) or four characters (Long int precision). An error will be indicated if the string length is not two (Short int) or four (Long int).
Use:	The CVT$% function provides the means to speed the processing of a large amount of packed data within a file. Converting the internal binary representation to an ASCII string is a less time-consuming process with CVT$% than the NUM$ function.

Example:

```
LIST
270  ! THIS PROGRAM READS THE INTEGERS FROM THE FILES
275  ! CREATED FOR PREVIOUS CVT%$ EXAMPLE
280  !
290  ! 1. FROM filea
300  DIM B% (10%)
310  OPEN "filea" AS FILE 1
320  FOR J%=1% TO 10%
321    GET #1,B$ COUNT 5
331    B%(J%)=VAL(B$)
350  NEXT J%
351  !
355  CLOSE 1
360  !
370  ! 2. FROM FILEB BY USING CVT$%
380  !
390  OPEN "FILEB" AS FILE 2
400  FOR J%=1% TO 10%
430    GET #2,B$ COUNT 2
440    B%(J%)=CVT$%(B$)
450  NEXT J%
460  CLOSE 2
470  END
*basic*
```

CVTF$

Function: Returns the four- or eight-character string representation of a floating point number depending on whether Single or Double precision was in effect.

Mode: Program/direct

Format: CVTF$(<n>)

Arguments: n is a Single or Double precision floating-point number.

Use: This function permits dense packing of floating point data in records. For example, any floating point number between 2.93874E-39 through 1.70141E+38 (single precision) can be stored in a four-character string and between 1.79769313486232E+308 through 4.4501477170144E-308 in an eight-character string (double precision).

Example:

```
LIST
   2  !  THIS PROGRAM STORES A FLOATING POINT ARRAY
   4  ! ON A DISC FILE IN A COMPACT FASHION
  10  DIM A(100)
1000  PREPARE "fil" AS FILE 1%
1010  FOR I% = 1% to 100%
1020  PUT #1%, CVTF$(A(I%))
1030  NEXT I%

1040  CLOSE 1%
*basic*
```

CVT$F

Function:	Returns the floating point number representation of the four- (Single precision) or eight-character (Double precision) string.
Mode:	Program/direct
Format:	CVT$F(<string>)
Arguments:	String is any string variable or constant having four characters (Single precision) or eight characters (Double precision). An error will be indicated if the string length is not four (Single) or eight (Double).
Use:	The CVT$F function provides the means to speed the processing of a large amount of packed data within a file. Converting the internal binary representation to an ASCII string is a less time-consuming process with CVT$'% than the NUM$ function.

Example:

```
LIST
   2  ! THIS PROGRAM READS BACK THE ARRAY
   3  ! CREATED BY EXAMPLE FOR CVTF$
   4  ! LEN (CVTF$(0)) IS USED TO DETERMINE IF
   5  ! SINGLE OR DOUBLE PRECISION IS USED
  10  DIM A(100):L%=LEN(CVTF$(0))
2000  OPEN "fil" AS FILE 1%
2010  FOR I% = 1% to 100%
2020  GET #1 A$ COUNT L% : A(I%) = CVT$F(A$)
2030  NEXT I%
2040  CLOSE 1%
*basic*
```

6.5 Miscellaneous functions

The following miscellaneous functions are described in this section:

Table 6-4 Miscellaneous Functions

Function	Description
CUR	Positions the cursor on specified line and column.
ERRCODE	Returns the value of the most recent error code.
FN<name>	Accesses a user-defined function.
TIME$	Returns year-month-day, hour.minutes.seconds.

The four Miscellaneus functions are described in detail in subsequent paragraphs.

6.5.1 Miscellaneus Functions

CUR

Function:	Moves the cursor to the specified row and column on the screen.
Mode:	Program/Direct
Format:	CUR(y%,x%)
Argument:	y% is the line where the cursor is to be moved with values of 0 to the maximum number of lines on the terminal used. x% is the position on the line with values of 0 to the maximum number of columns on the terminal used.
Result:	String.
Note:	CUR must be preceded on the program line by a ";" or "PRINT".
Use:	This function generates a string which, when printed, places the cursor at the specified row and column on the screen. The string returned is determinated from the terminal capability database 'termcap' as the entry "cm". (Refer to UNIX manual termcap (5).)

Examples:

```
Ex. 1
100  PRINT CUR(12,20) "BASIC III"
200  ; CUR(13,22) "REFERENCE MANUAL"
:
:

Ex. 2
10 PRINT CUR (10,10); "COLUMN 10, ROWN 10"
20 A$ = CUR (1,2) + 'ROW 1, COLUMN 2' : PRINT A$
```

ERRCODE

Function:	Returns the value of the latest generated error code.
Mode:	Direct/Program
Format:	ERRCODE
Use:	The ERRCODE function is normally used in conjunction with the IF and ON statements. If no error has been indicated the function value is 0.
Result:	Numeric

Example:

```
LIST
10  REM THIS PROGRAM WORKS FOR POSITIVE NUMBERS ONLY.
20  REM-----
30  ON ERROR GOTO 90
40  OPEN "CON:" AS FILE 0%
50  INPUT "POSITIVE NUMBER " A ;
60  Z=SQR(A)
70  PRINT "SQUARE-ROOT OF:" A "IS-->" Z
80  STOP
90  IF ERRCODE=142 THEN ; " NO NEGATIVE NUMBERS ALLOWED"
100 IF ERRCODE=210 THEN ; " NO CHARACTERS ALLOWED"
110 ; ERRCODE
120 ; CHR$(7) : REM BELL SOUNDS AFTER ERROR MESSAGE
125 REM IS PRINTED
130 RESUME 50
140 END
RUN
POSITIVE NUMBER? A  NO CHARACTERS ALLOWED
 210 (bell sounds)
POSITIVE NUMBER? -5  NO NEGATIVE NUMBERS ALLOWED
 142 (bell sounds)
POSITIVE NUMBER 9  SQUARE-ROOT of: 9 IS--> 3
:
:
```

FN

Function: Calls a user-defined function.

Mode: Direct/Program

Format: FN<name> [<type>] [(parameter)]

Arguments: Name is any valid variable name.
Type is optional and can be either % (or ".") or $.

Parameter consists of one or more variables and constants. Variables are passed to the defined function. They must be specified if they were included in the DEF FN statement.

Result: Depends on the type.

Use: This function allows the programmer to call a user-defined function in the same way as, for example, SIN(x) would be called.

Note: See DEF FN statement, on definition routines.

Examples: Ex. 1

```
EXTEND
LIST
5   REM **DEFINE AND USE SECANT FUNCTION***
10  DEF FNSEC(X)=1/SIN(X)
20  ;INT(FNSEC(PI/4%))
RUN
 1
*basic*
```

```
Ex. 2
LIST
 5  EXTEND
10  ! THIS EXAMPLE COMPUTES THE VOLUME OF A
20  ! SPHERE WITH RADIUS(R) IN RANGE: 1<=R<=4
50  DEF FNSPVOL
60    FOR R=1 TO 4
70      X=R**3
80      Y=PI*X
90      Z=4/3
100     VOL=Y*Z
110     ; FIX(VOL)
115   NEXT R
116   CLOSE
120   RETURN 0
130 FNEND
140 X=FNSPVOL
150 ; "END "
160 END
RUN
  4
 33
 113
 268
END
*basic*
```

TIME$

Function: Returns year-month-day and hour.minutes.seconds.

Mode: Program/direct.

Format: TIME$

Use: TIME$ reads the current time and date.

Result: String

Example:

```
10   ; TIME $
20   END
RUN
1984-06-01 10.15.30 (current time)
*basic*
```

Note: To set the time refer to DNIX (UNIX) manual date(1)

7. ADVANCED PROGRAMMING

Contents

7. ADVANCED PROGRAMMING

7.1 Introduction

This section contains information that should only be applied by user's who have a complete understanding of Basic III and the operating system. Subjects discussed here include advanced statements and functions as well as the use of Assembler routines from BASIC.

WARNING WARNING WARNING WARNING

Most of the advanced statements are hardware dependent and therefore will cause severe portability problems if the programs are to be moved to other systems. The user is recommended to avoid the use of these statements.

7.2 Advanced statements and functions

This section contains Basic III statements and functions which are to be used for advanced programming. The user is cautioned that if certain of these statements are used incorrectly, program execution may be inadvertently destroyed. The advanced programming statements and functions are summarized in Table 7-1.

Table 7-1. Advanced Programming Statements and Functions

Statement/ Function	Description
CALL	Calls an Assembler Program.
INP	Returns value of data from in-port specified.
OUT	Sends data to the out-port specified.
PEEK	Returns memory contents of a specified address (1 byte).
PEEK2	Reads the contents of two bytes.
PEEK4	Reads the contents of four bytes.
POKE	Changes or loads a value into specified address.
SYS( )	Provides essential system information.
VAROOT	Returns starting address of a table containing data about a variable.
VARPTR	Returns starting address of where a variable is stored.

The advanced statements and functions given in table 7-1 are described in detail in subsequent paragraphs.

7.2.1 Advanced statements and functions

CALL

Function: Calls an Assembly program and returns the contents of a CPU register. Which register is dependent on the processor type in your system.

Mode: Direct/Program.

Format: CALL(A%[,D%])

Arguments: A% is an integer holding the address of the machine code being called.

D% is optional integer parameter which will be pushed on the stack at the call.

Note: The assembly routine should always return to Basic III by executing a return instruction.
At return, the function will take the value in a CPU register. Which register is dependent on the processor type in your system.

Example:

```
LIST
5  ! DEFINE ADDRESS WHERE ASSEMBLY ROUTINE STARTS.
10  A%=1234%
15  ! DEFINE THE PARAMETER WHICH WILL BE TRANSFERRED
17  ! TO ROUTINE
20  D%=ASCII("A")
25  ! CALL THE ASSEMBLY ROUTINE AND PUT THE RETURNED
27  ! RESULT IN H%
30  H%=CALL(A%,D%)
40  END
```

Caution: This function is machine-oriented and should only be used for advanced programming. CALL can destroy a program execution if used erroneously.

INP

Function: Returns the value of data from the in-port specified.

Mode: Direct/Program

Format: INP(<channel no.>,<port>)

Use: Access to an in-port is accomplished through a channel, which has been opened with the OPEN statement. The <fd> in the OPEN statement should be the name of the special devicehandling INP, OUT functions in the system.

Example:

```
10  OPEN  "/dev/inp" AS FILE 1
20  A= INP(1, 32)
```

Note: Refer to D-NIX manual.

OUT

Function:	Sends data to the out-ports specified.
Mode:	Direct/Program
Format:	OUT #<channel no.> <port,data> [port,data...]
Arguments:	All parameters shall be specified in decimal. The different ports (interface card addresses), are found in the system manual.
Use:	This is a machine-oriented statement meant for advanced programming. The channel no.> is the channel which should have been opened with OPEN statement to the wanted device. See INP statement in this section. The statement, used in conjunction with the INP function gives the user access to the I/O-handling of the system.
Note:	The user should be familiar with the I/O-handling. Refer to system manuals for details.

PEEK

Function:	Returns the memory contents (of 1 byte) of a specified address.
Mode:	Direct/Program
Format:	PEEK(<address>)
Argument:	Address is the byte in memory to be accessed. It is specified in decimal.
Use:	PEEK is mainly used when Basic III works together with Assembler subroutines.

Example:

```
10  REM PEEK:----RETURNS THE CONTENTS OF THE MEMORY
20  REM AT THE GIVEN ADDRESS
30  FOR A%=: SYS(10) TO : SYS(10)+2%
40    PRINT A% TAB(10) PEEK(A%)
50  NEXT A%
60  END
RUN
 594      3
 595      3
 596      5
```

Note: The above example is for illustration purposes only. The result will vary depending on the memory contents of the locations displayed.

PEEK2

Function:	Reads the contents of two bytes.
Mode:	Direct/Program
Format:	PEEK2(<address>)
Argument:	Address is the starting byte in memory to be accessed.
Use:	PEEK2 is mainly used when Basic III works together with Assembler subroutines.

Example:

```
10 A% = PEEK2(SYS(10))
20 ;A%
RUN
-3763
*basic*
```

Note: The above example is for illustration purposes only. The result will vary depending on the memory contents of locations pointed to by SYS(10).

Note: If the specified address points to a memory location outside the users data area, one of the following system errors are displayed:

```
*basic* :Segmentetion error
or
*basic* :Bus error
```

PEEK4

Function:	Reads the contents of four bytes.
Mode:	Direct/Program
Format:	PEEK4 (<address>)
Argument:	Address is the starting byte in memory to be addressed.
Use:	PEEK4 is mainly used when Basic III works together with assembler subroutines.

Example:

```
10  A% = 2
20  B% = PEEK4(VARPTR(A%))
30  ; B%
RUN
2
*basic*
```

POKE

Function: Changes or loads a specific value into the designated address in RAM.

Mode: Direct/Program

Format: POKE <address>,<data> [,data,...]

Arguments: Address is the starting byte in memory where the data is to be loaded. It is specified in decimal.

Data is the decimal equivalent of the 8-bit binary number to be set.

If more than one DATA-value is given, the address is incremented one step for each new data value.

Use: Poke is mainly used when Basic III works together with assembler subroutines.

Caution: If POKE is used erroneously it may destroy the contents of needed memory locations.

Example:

```
10 DIM A%
20 A%=1%
30 ; VARPTR(A%),A% ! Addr > 32K will be neg
40 POKE VARPTR(A%)+3%,2
50 ; PEEK4( VARPTR(A%) )
60 ;A%
RUN
131915       0
 2
 2
*basic*
```

Note that the value '131915' of the VARPTR() is only an example.

The above example is the result on a CPU which does not do byte swapping (i.e. M68000). On other CPU's the result will be different.

Note: If the specified address points to a memory location outside the users data area, one of the following system errors are displayed:

Basic III :Segmentation error

Basic III :Bus error

SYS( )

Function: Provides essential system information.

Mode: DIRECT/PROGRAM

Format: SYS(i%)

Argument: i% can have the values shown below:

SYS(0)	- Reserved for future use, and will presently cause an error 143 (illegal sys function) if used.
SYS(1)	- Is reserved for future use, and will presently cause error 143 if used.
SYS(2)	- Returns total space available for program and data in the user partition. Is the default space plus the extra memory allocated at the start-up of BASIC with the basic -mXX option.
SYS(3)	- Gives current program code size.
SYS(4)	- Gives space left. This is a dynamic value, changing, depending on the allocated data variables.
SYS(5-7)	- Is reserved for future use, and will presently cause error 143 if used.
SYS(8)	- Is reserved for future use, and will presently cause an error 143 if used.
SYS(9)	- Reserved for future use, and will presently cause an error 143 if used.
SYS(10)	- Points to an information block about the internal variables for the BASIC interpreter.
SYS(11)	- Gives a pointer to lower memory limit for the stored program.
SYS(12)	- Gives a pointer to the variable root for the variable names in the user program.

NOTE! The SYS(12) should not be used. The two functions VARPTR() and VAROOT() are available for controlled access to the user variable pointers. The SYS(12) CAN NOT be used with programs which have been squeezed.

VAROOT/VARPTR

Function: VAROOT returns the base address of a table (or root) which contains information about a variable.

VARPTR returns the starting address where a variable is stored.

Mode: Program/Direct.

Format: VAROOT(<variable>)
VARPTR(<variable>)

Arguments: Variable can be a string variable or arrays of any kind (integer, string, or floating point).

Use: VARPTR can be use to locate the address of a variable. The variable can then be seen with PEEK and changed, if desired, via POKE.

VAROOT is use to change the actual length of a string variable. For string arrays, it can be used to find the address of a root table containing the addresses of where individual array elements are stored.

VARPTR can be used to access a machine code routine, stored in a character string. Note that the code must be address independant or use addresses, relative to a pointer, set up by BASIC before calling the routine.

Example: 1. Use of VARPTR and VAROOT for string variables.

```
                 VARPTR---------------,
                                      I
                   Table              I       Stored Variable
                 ,------------------, I      ,---------------,
VAROOT-->addr  n I Dimension length I '->  m I               I
                 I                  I        I               I
             n+2 I VARPTR address   I    m+1 I               I
                 I                  I        I               I
             n+6 I Actual string    I    m+2 I               I
                 I length           I        I               I
                 '------------------'    m+3 I               I
                                             I               I
                                             I               I
                                             I               I
```

```
10  DIM X$=10
20  X$="AB"
30  ;VARPTR (X$)
40  ;VAROOT (X$)
50  END
RUN
 2345
 1234
*basic*
```

The above example shows the use of VARPTR and VAROOT for string variables. Here VARPTR points to location 2345 where the string AB is stored.

```
                 ,--------------,        ,--------------,
address 1234 I        10        I 2345 I        A        I
             I                  I      I                 I
        1236 I       2345       I 2346 I        B        I
             I                  I      I                 I
        1240 I        2         I 2347 I                 I
             I                  I      I                 I
             '--------------'          I                 I
```

NOTE: This example is for illustration purposes
          only. Program lines 30 and 40 can give
          various results depending on where the
          variable resides.

Example: 2.   Use of VAROOT and VARPTR for arrays (integer,
                  single, double float)

```
                    VARPTR (A%(0%)) ------------,
                    VARPTR (A%(1%)) ----------, 1
                    VARPTR (A%(2%)) --------, 1 1
                                            1 1 1
                         Table              1 1 1     Variable
VAROOT(A%(X%))        ,----------------,    1 1 1 ,----------------,
    '------> addr n I Allocated space I    1 1 '->I                I
                    I for whole array.I   1 1    I   ARRAY         I
              n+2 I Address to firstI    1 1    I ELEMENTS        I
                    I element.        I    1 1    I                I
              n+6 I No. of          I    1 1    I                I
                    I subscripts.     I    1 1    I                I
              n+7 I Type:           I    1 1    I The size of an I
                    I 4=Integer       I    1 '--> I element dependsI
                    I 4=Single Float  I    1      I on whether it  I
                    I 6=String        I    1      I is an integer, I
                    I 8=Double Float  I    1      I single float   I
               ,--  I                 I    1      I or double.     I
               1  n+8 I Lower bound   I    1      I                I
               1      I               I    1      I                I
  One Block    1 n+12 I Size of       I    '---> I                I
  For Each    -1      I Subscripts    I          I integer=4 bytesI
  Subscript    1      I               I          I single=4 bytes,I
               1 n+16 I Reserved      I          I float          I
               '-     I               I          I double=8 bytes,I
                 n+20 I               I          I float          I
```

NOTE: VAROOT always points to the variable table
          independent of the index of the variable.

```
10  DIM A%(3%) ! ALLOCATED INTEGER ARRAY.
20  A%(1%)=10%
25  ! POINTER TO VARIABLE A%(0%)
30  ; VARPTR (A%(0%))
35  ! POINTER TO VARIABLE A%(1%)
40  ; VARPTR (A%(1%))
45  ! POINTER TO VARIABLE ROOT FOR WHOLE ARRAY.
50  ; VAROOT(A%(0%))
```

```
55  ! POINT AT SAME AS ABOVE.
60  ; VAROOT(A%(1%))
65  ! PRINT CONTENTS OF VARIABLE A%(1%)
70  ; PEEK4(VARPTR(A%(1%)))
75  ! PRINT TYPE AND NUMBER OF INDEX IN ARRAY.
80  ; PEEK2(VAROOT(A%(1%))+6%)
90  END
RUN
-178994
-178998
-178974
-178974
 10
 260
*basic*
```

NOTE: This example is for illustration purposes only. Program lines 30, 40, 50, and 60 can give various results depending on where the variable resides.

Example: 3. Use of VAROOT and VARPTR for string arrays.

```
                 VARPTR (A$(0%)) ---,  (= Str.Elem.Addr 0)
                 VARPTR (A$(1%))--, 1  (= Str.Elem.Addr 1)
                                  1 1
                    Base table    1 1        Root table
                                  1 1
VAROOT(A$(X%)) ,----------------, 1 1    ,--------------,
    '---->    nIAllocated space I 1 1   mIDimension     I
               Ifor whole array.I 1 1    ILength        I  One
            n+2IRoot table      I 1 1 m+2IString ElementI Block
               Iaddress.        I 1 1    Iaddress (0)   I For
            n+6INo. of          I 1 1 m+6IString Length I Each
               ISubscripts      I 1 1    I              I Element
            n+7IType:           I 1 1 m+8IDimension     I
               I4=Integer       I 1 1    ILength        I
               I4=Single Float  I 1 1m+10IString ElementI
               I6=String        I 1 1    Iaddress (1)   I
               I8=Double Float  I 1 1m+14IString Length I
               I                I 1 1    I              I
            n+8ILower Bound     I 1 1m+16I              I
               I                I 1 1    I              I
           n+12ISize of this    I 1 1    I              I
One Block      ISubscript       I 1 1    I              I
For Each       I                I 1 1
Subscript  n+16IReserved        I 1 '--->,--------------,
               I                I 1      I   Array      I
           n+20I                I 1      I   Element 0  I
               I                I 1      '--------------'
               I                I '----->,--------------,
               I                I        I   Array      I
               I                I        I   Element 1  I
                                         '--------------'
```

```
5   ! ALLOCATE A STRING ARRAY
10  DIM A$(3%)=10%
20  A$(1%)="A"
25  ! POINTER TO VARIABLE A$(0%)
30  ; VARPTR (A$(0%))
35  ! POINTER TO VARIABLE A$(1%)
40  ; VARPTR (A$(1%))
45  ! POINTER TO VARIABLE ROOT FOR WHOLE ARRAY
50  ; VAROOT (A$(0%))
55  ! POINT AT SAME AS ABOVE
60  ; VAROOT(A$(1%))
65  ! PRINT ASCII VALUE OF FIRST CHAR. IN A$(1%))
70  ; PEEK(VARPTR(A$(1%)))
75  ! PRINT ACTUAL LENGTH OF A$(1%)
80  ; PEEK2(PEEK4(VAROOT(A$(1%))+2%)+10%)
90  END
RUN
0
178984
178932
178932
65
1
*basic*
```

NOTE: This program is for illustration purposes only. Program lines 30, 40, 50, and 60 can give various results depending on where the variable reside.

VAROOT always points to the variable table independent of the index of the variable.

FIELD

Function: FIELD allows access to a string storage area through more than one string name.

Mode: Program/Direct

Format:
```
FIELD <string'var1> IN <string'var2>
     AT <integer1> COUNT <integer2>
```

Use: The string element address of <string'var1> is set to point into the string of <string'var2>. It points to character <integer1> in the string. Both the dimension length and actual length of <string'var1> is set to <integer2>.

If <string'var2> is not allocated or <string'var1> is allocated for separate use, error 134 is generated.

Error 134 is also caused by an attempt to point outside the space allocated for <string'var2>.

Example: Create a data record consisting of two fields:

```
 A name  15 characters
 A phone number  4 characters

 10 EXTEND
 20 DIM Datrec$= 19%
 30 FIELD Name$ IN Datrec$ AT 1% COUNT 15%
 40 FIELD Phone$ IN Datrec$ AT 16%
 45 Datrec$=SPACE$(19)      (COUNT 4%)
 50 LET Name$= "JAN-OLOF PERSSON"
 60 Phone$= CUT%$(13145)
 70 ; Name$
 80 ; LEFT$(Datrec$, 15)
 90 ; CUT$%(PHONE$)
100 ; CUT$%(RIGHT$(Datrec$,"))
RUN                           16%
JAN-OLOF PERSSON
JAN-OLOF PERSSON
131415
131415
*basic*
```

7.3 Use of assembler routines in BASIC

Due to the fact that the memory addressing is done with hardware (MMU, Memory Management Unit), the actual memory address where the program will be loaded in the main memory is not known.

The assembler code loaded must be PIC (Position Independent Code) which means that no absolute addresses are allowed to data or code.

When more than one parameter shall be passed to the Assembler program, the best way is to store the parameters in a vector, and pass the address of this vector to the routine.

7.4 Terminator characters

A data terminator character is RETURN (13 dec), LINEFEED (10 dec).

These values will act as data terminators in all operating modes, unless keyboard input is being analyzed by the user's program on a single byte basis (e.g., by use of the GET <string variable> statement in Basic III).

The use of one of these alternate terminating characters will have no effect on treatment of the data entry.

8. ISAM DATABASE OPTION UNDER UNIX

Contents

8. ISAM DATABASE OPTION

8.1 Introduction to the Index Sequential Access Method

ISAM, Indexed Sequential Access Method, is a technique, used for indexed access to large data file. It can be used for random access using a key string as search argument, or for sequential access using the index.

The data is divided into RECORDS. The record have a fixed, user definable length, and they are stored in a fixed record length file, the DATA file. Each data file has an ISAM index file associated to it and a STR structure information file.

The ISAM file may contain up to 10 indices into the data file. Each index has a symbolic name. It contains one KEY for each data record. The key consists of a key string, which also is a part of the data record, and a pointer to that record. The keys are ordered within the index file to form a B-tree structure. The sorting may be set to use a non-default method, with the sortorder.tab file.

All record pointers are logical and file referencies are symbolic. This means that the data and ISAM files may be copied and utilized on any random access device supported by the operating system.

The ISAM file is initialized by a utility program (isamin). After initiation, the ISAM, STR and DATA files are built by the user, using the BASIC ISAM statements. Since the index file trees are built in a well structured way, there is no need for time consuming reorganizations once the indices are established. The access times will always be at an optimum.

All storage is done in the form of strings. The record length is fixed and the key fields, used as index, must start at fixed positions within the record.

The CVT-functions are used to read or store binary data bytes from/to the record string.

The key index strings, which are stored in the ISAM file in addition to the data in the DATA file, can be sorted in six different ways. Either as pure character values or numerically, where the string is decoded as a BASIC variable in binary format.

Using default sorting of ASCII character (See app.A). The sort order is according to the ASCII values, with two exceptions:

1. A lower case character has the same value as the corresponding upper case character.

2. The Swedish characters 'A with a ring', A with two dots' and 'O with two dots' are sorted according to the Swedish standard. Compare section F.12. The sorting is thus:

```
                Upper case:  Å  Ä  Ö  lower case: å   ä   ö
   ASCII values are:         93 91 92             125 123 124
```

The ASCII sort order may be re-defined in the sortorder.tab/B file.

8.2 ISAM key formats

Five different formats are defined for the key string. They are:

0 Binary

This is a string of bytes of selectable length. The string is interpreted as an unsigned binary integer, with the most significant byte first. Sorting is according to the un-signed binary values of the string.

1 ASCII

This is a string of bytes of selectable length. The bytes are interpreted as 7-bit ASCII characters. The upper and lower case characters have the same value when sorting. The characters Å, Ä and Ö are sorted according to the Swedish standard. These characters has the ASCII values 93,91,92. For lower case the values are 125,123 and 124 decimal.

The sort order of the ASCII format may be set to a non-default value if the file sortorder.tab is changed.

2 Short integer

This is a two-byte string, holding a signed integer binary value, with the most significant byte first. Sorting is in numerical order. This format is compatible with the string generated by the Basic III function CVT%$(I%) if the integer precision is set to short int.

3 Long integer

This is a four-byte string, holding a signed integer binary value, with the most significant byte first. Sorting is in numerical order.

This format is compatible with the string generated by the Basic III function CVT%$(I%) if the integer precision is set to Long int (default).

4 Floating point (SINGLE))

This is a four-byte string, holding a single precision floating point binary value, with a format, compatible with BASIC and Pascal variables. Sorting is in numerical order. The program must be in the SINGLE precision mode. The function CVTF$(F) creates the four-byte string from a variable.

5 Floating point (DOUBLE)

This is an eight-byte string, holding a double precision floating point binary value, with a format compatible with the BASIC double precision variable. Sorting is in numerical order. The program must be in DOUBLE precision mode. The function CVTF$(F) creates the eight-byte string.

8.3 ISAM index file format

The ISAM file format is built on a B-tree concept. This concept makes it possible to maintain the search path through the tree at an optimum, in spite of insertions and deletions of key items.

The first record of an STR file is a header record. It contains information about the ISAM file and its associated DATA file.

An STR file may contain up to 10 separate indices with symbolic names. All information about the indices e.g. symbolic name, key type, key length and the B-tree root pointer is stored in the STR file header. The ISAM file contains one B-tree for each index.

```
ISAM file header:       FH   XH0  XH1  XH2  XH3  ..   0

Index root:             0    K0   K1   K2   K3        ..   0


                                             SP   KP   KS


Intermediate level:     FP   K0   K1   K2   K3        ..   0


Lowest level:           FP   K0   K1   K2   K3         ..   0


                                            KP   KS
```

FH	=	File header
XH	=	Index header
K	=	Key
SP	=	Son pointer
FP	=	Father pointer
KP	=	Key pointer (points to the data record)
KS	=	Key string

8.4 ISAM syntax and statements

The following special BASIC statements and task files are available for creation and direct access to a database, using the Index Sequential Access method. The ISAM option is linked into the BASIC task file.

Task files:
----

isamin	task file to create the ISAM and STR files and specify the associated data file. If the data file does not exist, it is created.
sortorder.tab	is a 128 bytes table, which defines the default ASCII sort order, when the Basic III task is loaded and started.

ISAM index and data file types:

Default file types are always used. When the task 'isamin' is executed, you are promted for a file name <fname>. The three files <fname>.ISM, <fname>.STR and <fname>.DAT will then be created.

8.4.1 sortorder.tab, sorting order table

The sort order for ASCII keys, may be re-defined by a 128-byte table, stored on the system volume under the name /usr/etc/sortorder.tab.

NOTE! IT IS ABSOLUTELY ESSENTIAL THAT THE SAME SORT ORDER DEFINITION IS USED WITH THE SAME DATA BASE AT ALL TIMES. OTHERWISE, THE INDEX FILE MAY BECOME UNUSABLE.

Each byte in the table corresponds to one position in the ASCII table (0 - 127). In this position a number (0 - 255) shall be stored. This number indicates the values for using when sorting the characters.

Example: The letter 'A' has the ASCII value 65 and is character number 65 in the sort order. The letter 'a' (lower case) has the ASCII value 97, but in the position 97 in the default table the number 65 is stored, which gives 'a' the same position in the sort order as 'A'.

8.5 ISAM statements

BASIC statements:

Statement	Description
ISAM DELETE	Deletes a record from an ISAM index file.
ISAM OPEN	Opens an ISAM index file and its associated data file.
ISAM READ	Accesses an ISAM data file.
ISAM UPDATE	Modifies an existing record in the data file associated with an ISAM index file.
ISAM WRITE	Enters a new record intc the data file and updates all indices in the index file.
CLOSE	Close an ISAM file and the associated data file.

8.5.1 ISAM Statements

ISAM DELETE

Function: Removes a particular record from an ISAM index file.

Mode:	Program/Direct.
Format:	ISAM DELETE #<channel no.>,<stringvar> The abbreviated form 'ISDL' may be used for 'ISAM DELETE'.
Arguments:	Channel no. corresponds to the internal number on which the file is opened.Valid channel numbers are 1 to 250. Stringvar is a string variable which must be identical to the record last read on that <channel no.>. If the record is not equal to the string, no operation is performed and an error message is given.
Use:	This statement removes the appropriate keys from a designated record in the ISAM file. The data record is enabled for re-use by a subsequent ISAM WRITE statement. Before an ISAM DELETE can be done the record must be ISAM READ.

Example:

```
10 ISAM OPEN "vol/ifile" AS FILE 1
20 ISAM READ #1,A$ INDEX "NAME" KEY "SMITH"
30 ISAM DELETE #1,A$
```

The first record with the key 'SMITH' in the index 'NAME' is read and deleted.

ISAM OPEN

Function:	Opens both the index and data files for ISAM access on a file-structured device (disk or diskette).
Mode:	Direct/Program.
Format:	ISAM OPEN <string expression> AS FILE <channel no.> The abbreviated form 'ISOP' may be used for 'ISAM OPEN'.
Arguments:	The syntax is as in the OPEN statement. The string expression corresponds to the external filed specification for the ISAM and Data files to be opened as follows:

- String Constant - "<fd>" where fd is the file descriptor as previously defined in Section 1.3, but without the file type.

or

- String Variable - A$

The channel no. after AS FILE must have an integer value corresponding to the internal channel number on which the field is opened. Valid channel numbers are 1 to 250.

The filename is a string literal specifying the name of the index file you want to open. The data file associated with this index file is automatically opened after the index file is specified.

The pathname is not required in the filename if you are opening a file in current directory.

Use: ISAM OPEN is the only method used to open an indexed file for ISAM access. Once this files is opened the data contained in the corresponding data file can be read, written, deleted and updated using the appropriate ISAM statement.

Note: Note that one ISAM OPEN statement opens both the index and data files.

Examples:

Ex. 1
```
10 REM vol/ifile IS THE ISAM INDEX FILE IN DIRECTORY
   VOL
20 ISAM OPEN "vol/index" AS FILE 1
```

Ex. 2
```
10 REM PROGRAM PROMPTS FOR ISAM FILE NAME
20 INPUT "ISAM FILE NAME? "A$
30 ISAM OPEN A$ AS FILE 1
```

ISAM READ

Function: Accesses by key or sequentially to records contained in the data file associated with an ISAM index file. Sequential access is according to the sort order.

Mode: Direct/Program.

Format:
```
ISAM READ #<channel no.>,<stringvar> [INDEX <stringa>]
[ [KEY <stringb>] [FIRST] [LAST] [NEXT] [PREVIOUS] ]
```

The abbreviated forms 'ISRD' and 'PREV' may be used for 'ISAM READ' and 'PREVIOUS'.

Arguments: The channel no. corresponds to the internal channel number on which the file is opened. Valid channel numbers are 1 to 250.

stringvar is any legal string variable, into which the record is read.

stringa is either a string expression or string variable which defines the name of the index that is to be used.

stringb is either a string expression or string variable which defines the search key within the index.

Note: The FIRST, LAST, NEXT, or PREVIOUS keyword can be used in place of [KEY stringb] to position the pointer to the first, last, next, or previous record in a particular index. That record is read without naming a particular key.

Use: The following rules are in effect for ISAM READ:

1. If the INDEX option (stringa) is missing or empty, the index last used is selected. If no index has been defined after the ISAM OPEN statement, the first index in the ISAM file is used.

2. If the KEY option (stringb) is missing or empty, the next key for the given index in the ISAM file is used. If no key has been specified on this index, the first key for the given index in the ISAM file is used. (Note that the optional keywords, if given, directs the keyword when the KEY option is missing). Only one position of one index is remembered for each channel.

3. If neither INDEX nor KEY options have been defined after the last ISAM OPEN statement, a sequential read is performed.

4. If it is the first read operation after ISAM OPEN, the first index is selected and the first record by that index is read, unless otherwise specified.

5. The KEY string may be a substring of the record key. In this case, the first record (by index), that contains the given key at the correct position, is read.

6. If duplicate keys are present in the index, the first record that contains the key given is read.

7. After a successful operation, the record pointer points to the defined data record and the record is read into the user string variable. The ISAM DELETE and ISAM UPDATE statements requires a previous ISAM READ to set the record pointer. After write/update, the position will be set on the affected key.

8. If a key sought for, is not found, the position will be set to the next sequential key of the index referenced. See the ISAM filepointer examples.

Examples: The followinf examples illustrate the various ways ISAM READ can be used.

```
Ex. 1
10  ISAM OPEN "vol/masts" AS FILE 1
20  ISAM READ #1, A$
    (Reads first index since INDEX option is missing)

Ex. 2
10  ISAM OPEN "vol/masts" AS FILE 1
20  ISAM READ #1, A$ INDEX "NAME"
or
10  I$="NAME"
20  ISAM READ #1,A$ INDEX I$
    (Reads first record by selected INDEX)

Ex. 3
10  ISAM OPEN "volmasts" AS FILE 1
20  ISAM READ #1,A$
30  WHILE -1
40    ISAM READ #1,A$
50    ; A$   ! PRINT THE RECORDS TO THE CONSOLE
60  WEND

Ex. 4
5   ISAM OPEN "vol/ifile" AS FILE 1
10  OPEN "PR:" AS FILE 2
20  WHILE -1
30    ISAM READ #1,A$
40    ; #2, A$ ! PRINT THE RECORDS ON THE PRINTER
50  WEND
    (Performs sequential read since both INDEX and
    KEY options are missing.)

Ex. 5
10  ISAM OPEN "vol/ifile" AS FILE 1
20  ISAM READ #1,A$ INDEX "NAME" KEY "SMITH"
    (Reads selected record by selected index -
    random access of a particular record.)

Ex. 6
10  ISAM OPEN "first" AS FILE 2
20  ISAM READ #2, A$ INDEX "SSNUM" LAST
    (Reads last record by key using the
    "SSNUM" index.)
:
50  ISAM READ #2, B$ PREVIOUS
    (Reads the 2nd from the last record using
    the "SSNUM" key.)
```

ISAM index file positioning examples:

File layout at start:

```
A    B    C    D    E    F    (EOF)
```

After an ISAM READ, key='Q' (NOT FOUND!) :

```
                    PREV      NEXT

A    B    C    D    E    F    (EOF)
```

After an ISAM WRITE 'C' (DUPLICATE ERROR) :

```
        PREV NEXT

A    B    C    D    E    F    (EOF)
```

After ISAM UPDATE 'C' to 'E' (DUPLICATE ERROR) :

```
    PREV     NEXT

A    B    C    D    E    F    (EOF)
```

After ISAM DELETE 'E' :

```
             PREV NEXT

A    B    C    D    F    (EOF)
```

After ISAM UPDATE 'C' to 'E' :

```
        PREV      NEXT

A    B    D    E    F    (EOF)
```

After ISAM WRITE 'C' :

```
   PREV      NEXT

 A    B    C    D    E    F    (EOF)
```

ISAM UPDATE

Function:	Alters an existing record in the data file and produces key changes to the index file when applicable.
Mode:	Program/Direct.
Format:	ISAM UPDATE #<channel no.>,<string1> to <string2> The abbreviated form 'ISUP' may be used instead of 'ISAM UPDATE'.
Arguments:	Channel no. corresponds to the internal channel number on which the file is opened. Valid channel numbers are 1 to 250. String1 is a string variable and must be identical to the record last read on that <channel no.>. If the record is not equal to string1, no operation will be performed and an error will be generated. String2 is a string variable and will replace string1 in the data file. All changed indices will be updated when this replacement occurs.

Use: Before using ISAM UPDATE, the appropriate file and records must be ISAM opened and ISAM read. If a duplicate key occurs in an index where it is not allowed, that index will not be updated, and an error will result. For example, if the name SMITH was used as a key for record 50 and you wanted to change record 20's key to SMITH,an error would result.

When a key positioned to, is deleted by ISAM UPDATE the position is moved to the new key inserted into the index referenced.

Example:

```
LIST
10 ISAM OPEN "vol/ifile" AS FILE 1
20 ISAM READ #1,A$ INDEX "NAME" KEY "SMITH"
30 B$="SMITH    NEW YORK    726-2677"
40 ISAM UPDATE #1,A$ to B$
*basic*
```

ISAM WRITE

Function: Enters a new record into the data file associated with an ISAM index file and adds the new keys in the index file.

Mode: Program/Direct.

Format: ISAM WRITE #<channel no.>,<stringvar>

The abbreviated form 'ISWR' may be used for the 'ISAM WRITE'.

Arguments: The channel no. corresponds to the internal channel number on which the file is opened. Valid channel numbers are 1 to 250.

Stringvar is any legal string variable, containing the entire new record.

Use: The record is appended to the data file and all indices are updated. The record must contain information in all key fields. If a duplicate key occurs in an index where it is not allowed, no operation will be performed and an error will be reported.

Examples:

```
Ex. 1
10  ISAM OPEN "vol/ifile" AS FILE 1
20  ISAM WRITE #1,"SMITH    NEW YORK   632-3256"

Ex. 2
10  ISAM OPEN "vol/ifile" AS FILE 1
20  A$="SMITH    NEW YORK    632-3256"
30  ISAM WRITE #1,A$
```

Position will be at the inserted key.

CLOSE

Function:	Closes both the ISAM index file and the associated data files.
Mode:	Program/Direct
Format:	CLOSE <n1[,n2,...]>
Arguments:	Channel no. is the internal channel number, on which the ISAM files has been opened.
Use:	An ISAM file should be closed before removing the disc/diskette from the disc/diskette station. Also the BASIC commands CLEAR and CLOSE close the ISAM files correctly.

Example:

```
10 ISAM OPEN "vol/ifile" AS FILE 1
:
:
90 CLOSE 1
```

8.6 Notes on ISAM performance

8.6.1 Disc space requirements

The size of the data files is determined by the number of records in the file and the data record length:

$$\text{Size(D)} = \text{Number(D)} * \text{Length(D)}$$

The size of the ISAM file is determined by the number of records in the data file, the number of indices used and the length of each index key string. Each key consists of a key string and a pointer. The space needed to accommodate all keys is:

$$\text{Size.calculated} = \text{Number(D)} * \sum_{i}^{NI} (KL(i) + KP + 1)$$

where	Number(D)	Is the number of data records
	KL(i)	Is the length of the index key string 'i'
	KP	Is the size of a key pointer
	NI	Is the number of indices, defined as keys.

However, due to the principle of B-trees, only an average of 75% (50% worst case) of the ISAM file will contain key information. So the actual size of the ISAM file may be approximated to:

$$\text{Size.average} = 4/3 * \text{Size.calculated}$$

or

$$\text{Size.worst case} = 2 * \text{Size.calculated}$$

8.6.2 ISAM Random access time

The time required to access a randomly selected data record, using ISAM depends on:

a) The height of the index tree
b) The size of the files used
c) The physical access time of the device

Principally, there is one random access required for each level of the index tree, plus one access to fetch the actual data. The number of levels in the index tree is determined by a number, called the 'order' of the tree, e.g. the number of keys stored in each record, and the total number of keys in the tree. The 'order' N of the tree is given by:

$$N = (\mathrm{INT})\ \frac{RI - FP - SP - 2}{KL(i) + KP + SP + 1}$$

The root record of the tree contains between 1 and N keys. All other records contains between N/2 and N keys. There is one extra Son pointer in all records. In worst case, the root contains one key, and all other records contain N/2 keys. In this case, the tree can be viewed as two sub-trees, which contains (Number(D) - 1)/2 keys each. Hence the height of the tree will be:

$$H <= 1 + \frac{\ln\left(\frac{\mathrm{Number}(D) - 1}{2}\right)}{\ln\ (\ N/2 + 1)} \qquad \text{(worst case)}$$

In the best case, the corresponding will be:

$$H >= \frac{\ln\ (\mathrm{Number}(D))}{\ln\ (N + 1)} \qquad \text{(best case)}$$

Thus, (H+1) random accesses are needed to read a data record by key selection.

The size of the file will affect the physical seek time on the device, and the amount of overhead to find the actual disc record within the logical file.

The seek time will be directly proportional to the average access time on the physical device.

The parameter values in the formulas are:

KP	= 4	(Key pointer)
RI	= 1024	(Record length of ISAM file, suggested value)
FP	= 4	(Father pointer)
SP	= 4	(Son pointer)
KL(i)	= The length of the index key string (i)	

Note: RI is specified when 'ismin' is executed.

11. QUICK REFERENCE SUMMARY

Reference & Format	Use	Page
ABS(x)	Returns absolute value of x.	6-2
ADD$(A$,B$,p%)	Returns the addition of two strings.	6-11
ASCII(A$)	Returns the ASCII value of first character of A$.	6-12
ATN(x)	Returns the arctangent (in radians) of x.	6-3
AUTO [lineno.][incr]	Automatic line numbering.	4-2
BYE	Transfers control to OS.	5-42
CALL(A$[,D%])	Calls an assembler program. CALL can destroy program execution if used erroneously.	7-2
CHAIN <"fd"> or CHAIN A$	Loads and executes a program.	5-43
CHR$(m1[,m2,m3,...])	Returns a character string corresponding to the ASCII values of the arguments.	6-12
CLEAR	Clears all variables and closes all open files.	4-3
CLOSE [channel no,..]	Closes the file(s).	5-15
CLS	Returns a string to clear the screen.	6-12
COMMON <list> COMMON <var$=ln> [list] (,string var,...)	Declares the variables, whose values are to be transferred to another program.	5-44
COMP%(A$,B$)	Returns a truth value based on a comparison two numeric strings.	6-13
CON (or CONT)	Continues program execution.	4-4
COS(x)	Returns the cosine of the x (x is in radians).	6-3
CUR(<y,x>)	Moves the cursor to line y%, position x%.	6-24
CVTF$(n)	Returns a four- or eight-character string representation of a floating point number.	6-22

Reference & Format	Use	Page
CVT$F(string)	Returns the floating point number representation of the first four or eight characters of a string.	6-23
CVT$%(string)	Returns the integer representation of the first two characters of a binary string.	6-21
CVT%$(integer var.)	Returns a two-character string representation of an integer.	6-20
DATA <list>	Assigns values to variables (used with READ).	5-2
DEF FN<name>[(argument)]=<function>	Defines a single line function.	5-45
DEF FN<name> (type)[arguments][LOCAL variable,variable,...]	Defines a multiple line function.	5-45
DIGITS <number>	Specifies the number of digits to be printed.	5-16
DIM <var list> or DIM <argument=expr>	Allocates space for strings and vectors.	5-3
DIV$(A$,B$,p%)	Returns the quotient A$/B$ rounded off to (+) p% decimals or to (-) p places of precision.	6-14
DOUBLE	Sets floating point numbers to double precision (16 digits) mode.	5-5
ED [line no.]	Starts program editing.	4-5
ELIF	Multiline IF statement else if clause.	5-53
ELSE	Multiline IF statement else clause.	5-53
END	Terminates the program.	5-48
ERASE <argument>	Erases one or more program lines.	4-6
ERRCODE	Returns the value of the latest generated error code.	6-24
EXP(x)	Returns the value ex.	6-3
EXTEND	Allows extended variable names to be used.	5-5
FIELD <str var> IN <str var> AT <int var> COUNT <int var>	Allows multiway access to string storage areas.	7-10

Reference & Format	Use	Page
FIX(x)	Returns the truncated value of x.	6-4
FLOAT	Specifies that all numbers will be interpreted as floating point.	5-6
FN<name> [type][(parameter)]	Calls a user defined function.	6-25
FNEND	Terminates a multiple line function.	5-49
FOR <var>=<expr> to <expr> [STEP expr]	Starts a program loop.	5-49
GET <string variable>	Reads one or more characters from the keyboard.	5-17
GET #<channel no.>,<string var> [COUNT number]	Reads from a file.	5-17
GOSUB <line no.>	Unconditional jump to a subroutine.	5-51
GOTO <line no.>	Unconditional jump to the given line number.	5-53
HEX$(x)	Returns the hexadecimal string representation of a decimal number.	6-4
IF <condition> THEN <argument1> [ELSE <argument2>]	Conditional control of the order of execution of the program lines.	5-55
IF-IFEND	Multiline IF statement.	5-53
INP(ch, i%)	Returns the data value from the in-port i%.	7-2
INPUT[argument]<list>	Fetches data for the current program.	5-18
INPUT LINE [#channel no.>,]<string variable>	Accepts a line of characters.	5-20
INSTR(n%,A$,B$)	Returns string B$ in A$ starting at position n%..	6-14
INT(x)	Returns the value of the greatest integer less than or equal to x.	6-4
INTEGER	Specifies that all variables are supposed to be integer variables, unless otherwise declared.	5-7
KILL <"fd">	Erases the file in question from external storage.	5-21

Reference & Format	Use	Page
LEFT$(A$,i%)	Returns the first i% characters of the string A$.	6-15
LEN(A$)	Returns the string length (including spaces) of A$.	6-15
[LET] <var> = <expr>	Assigns a value to a variable.	5-8
LIST [argument]	Lists, saves or prints a program.	4-7
LOAD <fd>	Loads a program into working storage of the computer.	4-8
LOG(x)	Returns the natural logarithm of x.	6-5
LOG10(x)	Returns the common logarithm of x.	6-6
LONG INT	Sets the integer precision to long (4 bytes) integers (default). This only effects the CVT%$ AND CVT$% functions.	5-9
MERGE <fd>	Merges program files.	4-9
MID$(A$,p%,k%)	Returns the substring of A$, which starts in position p% and has a length of k% characters.	6-15
MOD(<argument1>,<argument2>)	Returns the remainder of an integer division of the arguments.	6-6
MUL$(A$,BS,p%)	Returns the product A$*B$ with p% (+) decimals or with p (-) places of precision.	6-16
NAME <'fd1"> AS <"fd2>	Changes the name of a file.	5-22
NEW	Clears storage.	4-10
NEXT <variable>	NEXT terminates a program loop, which begins with a FOR statement.	5-57
NO EXTEND	Terminates work in EXTEND mode.	5-9
NO TRACE	Terminates the printout of line numbers, which was started by the instruction TRACE.	5-58
NUM$(argument)	Returns the numeric string corresponding to the argument.	6-17
OCT$(argument)	Returns an octal string representation of a decimal number.	6-6

Reference & Format	Use	Page
ON ERROR GOTO <line number>	Branches to the indicated line number of an error.	5-58
ON <expression> GOSUB <line no.> [,line no.,...]	Conditional jump to one of several subroutines or to one of several entry points in a subroutine.	5-60
ON <expr> GOTO <line no.>[,line no.,...]	Jump to one of several line numbers, depending on the value of the expression.	5-61
ON <expr> RESTORE <line no.> [,line no.,...]	Sets the DATA pointer by the same selection routine as ON - GOTO.	5-61
ON <expr> RESUME <line no.> [,line no.,...]	Jump to one of several line numbers, depending on the value of the expression. The error handling is resumed. Used with ON ERROR GOTO.	5-62
OPEN <"fd"> AS FILE <number>	Opens a file.	5-22
OPTION BASE <n>	Denotes the lowest vector index value	5-10
OPTION EUROPE <n>	Specifies European or American PRINT USING "," and "." field parameters.	5-25
OUT # <channel no.> <port,data> [port,data,...]	Addresses the out ports at data output.	7-3
PEEK(<i%>)	Returns the contents of one byte at storage address i%.	7-3
PEEK2(<b%>)	Returns the contents of two bytes at storage address b%. This function is meant for advanced programming.	7-4
PEEK4(<b%>)	Returns the contents of four bytes at storage address b%. This function is meant for advanced programming.	7-4
PI	Returns a constant value 3.14159 (single precision).	6-7
POKE <addrdata>[,data,..]	Loads a value into a storage cell.	7-4
POSIT #<channel no.>[,number]	Positions the file pointer.	5-26

Reference & Format	Use	Page
POSIT (<channel no.>)	Returns position of file pointer.	5-26
PREPARE <"fd"> AS FILE <number>	Creates and opens a new file.	5-28
PRINT [#channel no.,]<argument>[,argument,...]	Prints data in ASCII format.	5-29
PRINT [#Chno,] USING <"format string"><argument>[,argument,"]	Prints numbers and strings with the specified format.	5-32
PUT #<channel no.>,<string variable>	Writes a string variable in binary format.	5-39
RANDOMIZE	Sets a random starting value for the RND function (the random number generator).	5-10
READ <variable>(,variable,..)	Used together with DATA statements as a way of assigning values to variables.	5-12
REM text	Inserts comments in a program.	5-62
REN [<line no.1>[,incr[line no.2-line no.3]]]	Changes the line numbering of the current program.	4-11
REPEAT	Begins a loop terminated by UNTIL.	5-63
RESTORE <line number>	Makes possible renewed use of the contents of DATA statements.	5-12
RESUME <line number>	Returns from error handler.	5-64
RETURN <variable>	Returns from subroutine or multiple line function.	5-64
RIGHT$(A$,n%)	Returns the last characters of A$ starting at position n%.	6-17
RND	Returns a random number between 0 and 0.999999.	6-7
RUN <fd>	Loads and executes a BASIC III program or executes the current program.	4-12
SAVE <fd>	Creates a disk file and stores the current program into that file.	4-13

Reference & Format	Use	Page
SCR	Clears storage.	4-13
SGN(x)	Returns the value +1 if x positive, 0 if x and -1 if x negative.	6-8
SHORT INT	Sets the integer precision to short (2-byte) integers. This only effects the CVT%$ and CVT$% functions.	5-13
SIN(x)	Returns the Sine of x (x is in radians).	6-8
SINGLE	Changes variables and expressions, which are floating point numbers, to single precision (6 digits).	5-14
SLEEP <expr>	Suspends currently running program for a specified number of seconds.	5-65
SPACE$(n%)	Returns a string consisting of n% spaces.	6-18
SQR(x)	Returns the square root of x.	6-8
STAT	Gives mode and program size information.	4-14
STOP	Stops the program execution.	5-66
STRING$(I%,K%)	Returns a string of ASCII characters.	6-18
SUB$(A$,B$,p%)	Returns the arithmetic difference A$-B$ of the numeric strings A$ and B$ with (+) p% decimals or with (-) p places of precision.	6-18
SWAP%(n%)	Returns integer with first and the second bytes of n% transposed.	6-9
SWAP2%(n%)	Returns integer with bits 0-15 and bits 16-31 transposed.	6-9
SYS(i%)	Returns system status as follows:	7-5
SYS(2)	Returns total space available for program and data.	
SYS(3)	Program code size.	
SYS(4)	Remaining storage space.	
SYS(10)	Points to information block about the program.	

Reference & Format	Use	Page
SYS(11)	Starting address of the program.	
SYS(12)	Gives a pointer to the variable root for all user variables in DataBoard BASIC. Note. Not available in a squeezed program.	
TAB(i%)	Tabulates to the i%-th position on the line.	5-31
TAN(x)	Returns the tangent of x (x in radians).	6-10
TIME$	Returns year-month-day hours.min.sec	6-26
TRACE [#channel no.]	Prints the line number of the executed program lines.	5-66
UNSAVE <fd>	Erases a file from a disk.	4-14
UNTIL <expression>	Terminates a loop that begins with REPEAT.	5-67
VAL(A$)	Returns the numeric value of the numeric string A$.	6-19
VAROOT (variable)	Returns the address of a table, which contains information about a variable.	7-6
VARPTR (variable)	Returns the address of the value of a variable.	7-6
WEND	WEND terminates a loop that begins with WHILE.	5-67
WHILE <expression>	Specifies the condition for branching out of a program loop.	5-67

APPENDICES

Contents

APPENDIX A

A. BASIC III ASCII CHARACTER SET

Oct.	Hex.	Dec.	Char.	Oct.	Hex.	Dec.	Char.	Oct.	Hex.	Dec.	Char.	Oct.	Hex.	Dec.	Char.
000	00	0	NUL (CTRL Ä)	040	20	32	SPACE	100	40	64	É (@)	140	60	96	é (')
001	01	1	SOH (CTRL a)	041	21	33	!	101	41	65	A	141	61	97	a
002	02	2	STX (CTRL b)	042	22	34	"	102	42	66	B	142	62	98	b
003	03	3	ETX (CTRL c)	043	23	35	#	103	43	67	C	143	63	99	c
004	04	4	EOT (CTRL d)	044	24	36	¤ ($)	104	44	68	D	144	64	100	d
005	05	5	ENQ (CTRL e)	045	25	37	%	105	45	69	E	145	65	101	e
006	06	6	ACK (CTRL f)	046	26	38	&	106	46	70	F	146	66	102	f
007	07	7	BEL (CTRL g)	047	27	39	'	107	47	71	G	147	67	103	g
010	08	8	BS (CTRL h)	050	28	40	(	110	48	72	H	150	68	104	h
011	09	9	TAB	051	29	41	)	111	49	73	I	151	69	105	i
012	0A	10	LF	052	2A	42	*	112	4A	74	J	152	6A	106	j
013	0B	11	VT (CTRL k)	053	2B	43	+	113	4B	75	K	153	6B	107	k
014	0C	12	FF (CTRL l)	054	2C	44	,	114	4C	76	L	154	6C	108	l
015	0D	13	CR	055	2D	45	-	115	4D	77	M	155	6D	109	m
016	0E	14	SO (CTRL n)	056	2E	46	.	116	4E	78	N	156	6E	110	n
017	0F	15	SI (CTRL o)	057	2F	47	/	117	4F	79	O	157	6F	111	o
020	10	16	DLE (CTRL p)	060	30	48	0	120	50	80	P	160	70	112	p
021	11	17	DC1 (CTRL q)	061	31	49	1	121	51	81	Q	161	71	113	q
022	12	18	DC2 (CTRL r)	062	32	50	2	122	52	82	R	162	72	114	r
023	13	19	DC3 (CTRL s)	063	33	51	3	123	53	83	S	163	73	115	s
024	14	20	DC4 (CTRL t)	064	34	52	4	124	54	84	T	164	74	116	t
025	15	21	NAK (CTRL u)	065	35	53	5	125	55	85	U	165	75	117	u
026	16	22	SYN (CTRL v)	066	36	54	6	126	56	86	V	166	76	118	v
027	17	23	FTB (CTRL w)	067	37	55	7	127	57	87	W	167	77	119	w
030	18	24	CAN (CTRL x)	070	38	56	8	130	58	88	X	170	78	120	x
031	19	25	EM (CRTL y)	071	39	57	9	131	59	89	Y	171	79	121	y
032	1A	26	SUB (CTRL z)	072	3A	58	:	132	5A	90	Z	172	7A	122	z
033	1B	27	ESC	073	3B	59	;	133	5B	91	Ä ([)	173	7B	123	ä ({)
034	1C	28	FS (CTRL /)	074	3C	60	<	134	5C	92	ö (\)	174	7C	124	ö (¦)
035	1D	29	GS (CTRL ()	075	3D	61	=	135	5D	93	Å (])	175	7D	125	å (})
036	1E	30	RS (CTRL ü)	076	3E	62	>	136	5E	94	ü (^)	176	7E	126	ü (~)
037	1F	31	US (CTRL ')	077	3F	63	?	137	5F	95	_	177	7F	127	DEL

APPENDIX B

B. BASIC III ERROR MESSAGES

Table B-1 lists the error messages that can be returned when using Basic III. If an error is found during execution of a program the line number where the error occurred will be appended to the message.

Note that error messages from the operating system is reported with the error numbers according to the operating system.

The text messages are taken from the file /usr/etc/basicerr.txt. If this file is not available at the start of the BASIC task, only the error numbers are reported.

The /usr/etc/basicerr.txt file is delivered together with the BASIC interpreter task file and the BASIC program BASERRGEN/B, which may be used to generate a new BASICERR/A file on any volume specified by the user.

Table B-1. Error Messages

Number	Message	Meaning
1	Not owner	
2	Not such file or directory	
3	No such process	
4	Interrupted sysytem call	
5	I/O error	
6	No such device or address	
7	Arg list too long	
8	Exec format error	
9	Bad file number	
10	No children	
11	No more processes	
12	Not enough core	
13	Permission denied	
14	Bad address	
15	Block device required	
16	Device busy	
17	File exists	
18	Cross-device link	
19	No such device	
20	Not a directory	
21	Is a directory	
22	Invalid argument	
23	File table overflow	
24	Too many open files	
25	Not a typewriter	
26	Text file busy	
27	File too large	
28	No space left on device	
29	Illegal seek	
30	Read-only file system	
31	Too many links	
32	Broken pipe	
33	Argument too large	
34	Result too large	
35	Structure needs cleaning	

Number	Message	Meaning
36	Would deadlock	
37	Not a semaphore	
38	Not available	
39	File write protected	
40	File delete protected	
41	Disk full	
42	Disk not ready	
43	Disk write protected	
44	Logic file not opened	
45	Wrong logic file number	
46	Wrong unit number	
47	Wrong trap number	
48	Error in library	
49	Wrong fysical file number	
50	End of file	
51	Too long line	
58	Illegal character	
64	Illegal NAME	
68	Illegal time specification	
120	ISAM: Can't find key	
121	ISAM: Multiple key not allowed	
122	ISAM: Wrong key	
123	ISAM: Error on check-read	
124	ISAM: Can't find index	
125	ISAM: Wrong data record length	
126	ISAM: Wrong version of ISAM file	
127	ISAM:	
128	ISAM: End of memory	
129	ISAM:	
130	Floating point overflow.	Certain value out of range.
131	Array Index outside legal range	
132	Integer overflow	
133	Error in ASCII-arithmetic expression	
134	String index neg. or too large	
135	Negative TAB,SPACE$,STRING$ arg.	
136	Overflow in string assign	The dimensions of the receiving string are too small.
137	Attempt to expand array or string	A vector cannot be extended beyond its original length.
138	Expression out of range in ON	
139	RETURN without GOSUB	A return statement is encountered when no GOSUB has been executed.
140	Wrong return type	
141	Out of DATA statements	The data list is exhausted and a READ statement wants more data.
142	Wrong arg. to built-in function	
143	Illegal SYS function	
144	Illegal line	
145	FNEND without previous RETURN	

Number	Message	Meaning
146	PRINT USING error	Wrong format in PRINT USING statement.
147	Illegal data	
148	Too few data	
149	RESTORE not to DATA line.	
150	Too many data.	
151	RESUME without error.	
152	Attempt to read from outpipe or write to inpipe	
153	Open with untranslateable ':' expression	
176	Graphic point outside screen	
180	Can't find line	Reference to a nonexistent line number
181	Illegal GOTO into/out of function	
182	NEXT, WEND, IFEND or UNTIL missing	
183	FOR or WHILE missing	
184	Wrong variable in NEXT	
185	Illegal FOR, WHILE, REPEAT or mult in IF nesting	
186	FOR loop with local not allowed (use WHILE)	Use of the FOR loop with a local variable is not permitted. This applies to multiple line function
187	Function not defined	Call for undefined function
188	Several functions with same name	
189	Illegal function	Mixing of several DEF instructions is not allowed
190	Wrong number of indices	The number of indexes is not in accordance with the DIM statement
191	Not assignable in function	
192	REPEAT missing	
193	IF missing (multiline type)	
194	Multiple ELSE not allowed	
200	Unit not connected	
201	End of memory	
202	Program LIST-protected	
203	Illegal program format	
204	Attempt to MERGE .bac file	
205	COMMON must be first; CHAIN error	
206	Use RUN	
207	Can't continue	Applies to GOTO line number and CON.
208	Not allowed as command	
209	Wrong data to command	Wrong argument to the command e.g. LIST ##
210	Illegal number	The number contains other characters than digits
211	Precision can't be changed now	Change of precision after assignment not allowed
212	Compiler buffer overflow	
220	Don't understand	

Number	Message	Meaning
221	Illegal character after statement	
222	Must be first on a line	
223	Wrong number or types of argument	
224	Illegal mix of number and strings	
225	Not simple variable, index not allowed	
226	Illegal statement after ON.	Formal Basic III error
227	"," missing	
228	'=' missing	
229	')' missing	
230	"AS FILE" missing	Missing in OPEN and PREPARE instructions
231	"AS" missing	Error in NAME ...AS ... .
232	"TO" missing	In FOR loops
233	Missing line number	
234	Illegal variable	
235	# missing	
236	IN, AT or COUNT missing	

APPENDIX D

D. BASIC III --- BASIC vers.5 DIFFERENCE

Basic III differs to a large extent from the earlier BASIC versions. Below is a list of the MAIN differencies, describing the BASIC III characteristics compared to BASIC vers.5:

1 : The format of BASIC III programs in compressed form (SAVE) are not compatible with earlier Basic versions. Use files in uncompressed (LIST) form for compatibility.

2 : There is only one file type, byte oriented files. Extensive file handling is available from Basic III.

3 : Single step is obtained with character CTRL-Z (NOT CTRL-S). This is due to the fact that CTRL-S in most applications stops the output to the terminal.

4 : Integer size is 4 bytes, allowing a larger range of values.

5 : The integer arithmetic uses the microprocessor instructions (if available) for functions like \, *, + -. This may cause unexpected results. Example: The division (-3%)/2% migt give the result -2% or -1% depending on the inplementation of the instruction DIV.

6 : The floating point format is IEEE giving 6 digits in SINGLE and 15 in DOUBLE precision.

7 : The command STAT has been added to give information about the interpreter mode and the program and data sizes.

8 : New functions are:

PEEK4%(x)	Reads four bytes from address x.
SWAP2%(X)	Returns the first and second word in integer x transposed.
CLS	Returns a string to clear the screen.

9 : New statements are

Multiline IF	IF-THEN-ELIF-THEN-ELSE-IFEND
REPEAT UNTIL	Loop construct with condition at the end.
FIELD	For database applications.

10: The statements
SHORT INT
LONG INT
To control the integer to string convert functions CVT%$ and CVT$%.

11: The editor control keys are read from the file bascap or /usr/etc/bascap.

12: SYS(5), SYS(6), SYS(7) are not used.

APPENDIX F

F. AVAILABLE BASIC VERSIONS AND OPTIONS

The different versions of Basic III available are:

1. BASIC III ISAM D-NIX operating system.

   This is the main version,described in this manual. For several users the interpreter code is shared and every user gets a separate data area.

2. BASIC III ISAM for D-NIX operating system.

   The above version but with the single user ISAM included. (ISAM is described in Appendix F.)

3. Run-only BASIC versions are available on request, which does not include the interactive command interpreter.

NAME
basic - basic programming language interpreter

SYNOPSIS
```
basic [-d -x -i -c -m] [file]
```

DESCRIPTION Basic III Ver. 1.07

Basic is an interpreter for the basic dialect from Dataindustrier DIAB AB, Sweden.

The input editor determines the terminal capabilities from the terminal capability database termcap(5). The shell variable TERM should be set to the terminal type used.

The control characters or sequences the user wants to use for cursor motion, enter insert mode ... , can be redefined in the file '/usr/etc/bascap' using the same syntax as in termcap.

The entries read from termcap and bascap are:

Abrev.	termcap	bascap	default	meaning
"kr"	x	x	^L	right arrow key
"kl"	x	x	^H	left arrow key
"ku"	x	x	^K	up arrow key
"kd"	x	x	^N	down arrow key
"cm"	x			cursor positioning
"cl"	x			clear display
"ic"		x	^I	enter insert mode
"ei"		x	^U	exit insert mode
"dc"		x	^E	delete character
"cd"		x	^D	delete to EOL
"dl"		x	^X	delete line
"nd"	x			non destructive space
"up"	x			move cursor up
"co"	x		80	# of columns

The character '^' stands for the CTRL-key i. e. ^K is the same as pressing the CTRL-key and (while keeping it depressed) press K.

The arrow key entries in bascap overrules the ones in termcap if specified in both.

The options alters the mode of the basic interpreter when started

-d	double floating point precision
-x	extend mode (long variable names)
-i	integer mode
-c	disables CTRL-C and CTRL-Z function
-mXX	XX number of KBytes extra memory (default 32K)

If 'file' is specified the file 'file' is loaded and executed by the basic.

From the basic shell commands can be executed by starting the line with an exclamation mark (!). The rest of the line is then passed to a new shell which executes it.

NEW FEATURES

Additions in Basic compared to BasicII are:

ARGC%

Function:

Returns the number of parameters the basic interpreter was started with.

Mode: Direct/Program

Format: ARGC%

Result: Numeric

Use: ARGC% is used in combination with the function ARGV$ to retrieve the startup parameters.

Example:

Ex. 1
If the basic was started:

```
basic -x format fil1 fil2
```

ARGC% will return 5, since there are five space separated strings in the start command.

```
ARGV$(1) will return 'basic' ( the 5 character string: basic)
ARGV$(2)             '-x'
ARGV$(3)             'format'
ARGV$(4)             'fil1'
ARGV$(5)             'fil2'
```

ARGV$

Function:

Returns the parameters the basic interpreter was started with.

Mode: Direct/Program

Format: ARGV$(x)

Argument:

x is a integer number between 1 and the value returned by the function ARGC%.

Result: String

Use: ARGV$ is used in combination with the function ARGC% to
 retrieve the startup parameters.

Example:

Ex. 1
  If the basic was started:

```
  basic -x format fil1 fil2
```

ARGC% will return 5, since there are five space separated
strings in the start command.

```
ARGV$(1) will return 'basic' ( the 5 character string: basic)
ARGV$(2)             '-x'
ARGV$(3)             'format'
ARGV$(4)             'fil1'
ARGV$(5)             'fil2'
```

FIELD

format:

```
FIELD strvar1 IN strvar2 AT pos COUNT len
```

A statement for advanced programming, making it
possible to access a string storage area via
different string variable names.
The string variable 'strvar2' must have been allo-
cated (done with assignment or DIM) and 'strvar1'
should NOT have been allocated or allready be a
field in 'strvar2' otherwise there is an error.
The statement changes the 'strvar1' pointer to
point into the storage area for 'strvar2' at char-
acter 'pos' and sets the allocated length and
actual length to 'len'.
Usefull for data base applications (ISAM) to create
data records.

LOCK

Function: Test and lock a region of a file for
  exclusive use.

Mode: Direct/Program

Format: LOCK #<channel no.>,<count>

Arguments: Channel no. corresponds to the internal channel
  number on which the file is opened.

  Count is the number of bytes from the current
  file position that is to be locked. A positive
  value means 'count' bytes forward from current
  file position, a negative backwards from current
  position (not including current).
  Count "0" is used when the region from current

position to the end-of-file (present or future) should be locked.

Use: LOCK can be used when several processes access common files and want to ensure exclusive use of the file.

LOCK tests the specified region of the file, if it is already locked by another process an error is generated, otherwise the region is locked.

The lock is released when the statement UNLOCK is executed or the file is closed with CLOSE.

Note: If one process has opened a file and locked it with LOCK another process can still access the file unless it also tries to LOCK the same region or the file has the enforcement flag enabled.

For information on how to set the enforcement flag for a file, refer to the operating system manual lockf(2) and chmod(1).

Example: Ex. 1

```
10 OPEN "lfile" AS FILE 1
20 LOCK #1,20
```

The first 20 bytes of file 'lfile' is locked for exclusive use.

Ex. 2

```
10 OPEN "lfile" AS FILE 1
20 POSIT #1,20
30 LOCK #1,-5
```

Bytes number 15 to 19 are locked for exclusive use.

Ex. 3

```
10 OPEN "lfile" AS FILE 1
20 LOCK #1,0
30 PUT #1, "This is my file"
40 UNLOCK #1
```

The entire file is locked for exclusive use. A message is written to the file and the file lock is released.

UNLOCK

Function: Releases all region locks on a file which has been locked for exclusive use by a previous LOCK statement.

Mode: Direct/Program

Format: UNLOCK #<channel no.>

Arguments: Channel no. corresponds to the internal channel number on which the file is opened.

Use: UNLOCK is used to release the file lock(s) on a file. The locks are established with the statement LOCK to ensure exclusive use of the file.

The lock is also released when the statement UNLOCK is executed or the file is closed with CLOSE.

Example: Ex. 1

```
10 OPEN "lfile" AS FILE 1
20 LOCK #1,0
30 PUT #1, "This is my file"
40 UNLOCK #1
```

The entire file is locked for exclusive use. A message is written to the file and the file lock is released.

REPEAT - UNTIL

```
 format:
REPEAT
  statement(s)
UNTIL condition
```

Loops until condition is true.

IF - THEN - ELSE - IFEND, ELIF THEN:

```
 format:
IF condition [ THEN [:]]
  statement(s)
[ ELIF condition [ THEN ]
  statement(s) ]
[ ELSE
  statement(s) ]
IFEND
```

A multiline IF construct is distinguished from an ordinary IF by leaving the line empty after the condition, after THEN or by placing a colon after THEN.

In one multiline IF construct only one block of statements, at the most will be executed.
The construct must be terminated by an IFEND.
The ELIF construct can be repeated as many times as wanted.

SYSTEM

format: SYSTEM strvar

The contents of the string variable is passed to a subshell and executed as a command.

Example:

```
SYSTEM "date"
```

result:

```
Mon Feb 18 09:21:44  GMT+1:00 1985
```

PREPARE

format:

```
PREPARE strvar AS FILE fnr MODE mode
```

The protection of a prepared file is determined by the bit mask 'mode'. The same mask as for the system call creat(2) is used.
The default mode is 0644 (octal.) giving read and write permission to owner and read to group and others.
The prepared file is opened for READ and WRITE see also OPEN (mode).

OPEN

OPEN (mode)

format:

```
OPEN strvar AS FILE fnr MODE mode
```

Opening a file can be done in three modes

mode	meaning
0	READ
1	WRITE
2	READ and WRITE

Default is mode 2, READ and WRITE.

OPEN (pipes)

format:

```
OPEN "PIPEIN:cmd" AS FILE file number
or
OPEN "PIPEOUT:cmd" AS FILE file number
```

The shell command cmd is executed by a subshell with it's standard input (PIPEOUT) or standard output (PIPEIN) connected to the basic via the file referenced by file number.

OPEN (string replacement)

format:
```
OPEN "str1:str2" AS FILE file number
```

In order to make it possible to use old (BasicII) device names i. e. "PR:" a string replacement function has been implemented.
In the file translate.txt in current directory or, if it does not exist, in /usr/etc/translate.txt two lines specifying str1 and the wanted replacement string should be entered separated with line feed.

Example:

In translate.txt
```
PR:
PIPEOUT:print
```

The statement OPEN "PR:" AS FILE 1 will be translated to OPEN "PIPEOUT:print" AS FILE 1

SHORT INT/LONG INT

format:
```
SHORT INT
LONG INT
```

Specifies the integer size to use when using the string to integer and integer to string functions CVT%$ and CVT$%.
The default is LONG corresponding to 4 bytes, SHORT is 2 bytes.

OPTION EUROPE

format:
```
OPTION EUROPE n
```

Specifies the use of periods, commas and space in "PRINT USING" output.
OPTION EUROPE 0 gives period as decimal character and commas as separator characters.
OPTION EUROPE 1 gives comma as decimal character and periods as separator characters.
Added :
OPTION EUROPE 2 gives period as decimal character and spaces as separator characters.
This is default and conforms to "PRINT USING" in ABC800.

CLS

format:
```
PRINT CLS
```

String function returning a string to clear the the display of current terminal.

The string is the "cl" entry in termcap.

DNIX (integer function)

format:
R=DNIX(par1,par2, ...)

Operating system call.
The given parameters are pushed on the system stack to a total of 10 parameters and entry into the operating system is done with assembler instruction TRAP 4.
If less than 10 parameters are supplied the remaining ones are set to zero by Basic III.
The function returns the return value from the operating system.

REQUEST

Function:

Call to operating system.

Mode: Direct/Program

Format: R=REQUEST(req.code,par1,par2, ...)

Arguments:

Req.code determines the system routine called and par1, par2 and so on are sent as parameters to this routine. If less than 9 parameters are supplied the the remaining ones are set to zero.

Use:

Makes all operating system calls available.
The given parameters are pushed on the system stack to a total of 10 parameters and the specified operating system routine is called.

The function returns the return value from the operating system.

Req. code	Routine	Req. code	Routine
1	exit	30	utime
2	fork	31	stty
3	read	32	gtty
4	write	33	access
5	open	34	nice
6	close	35	ftime
7	wait	36	sync
8	creat	37	kill
9	link		
10	unlink	41	dup
11	execv	42	pipe

12	chdir	43	times
13	time	44	profil
14	mknod		
15	chmod	46	setgid
16	chown	47	getgid
17	brk	48	signal
18	stat		
19	lseek	54	ioctl
20	getpid		
21	mount	59	execve
22	umount	60	umask
23	setuid	61	chroot
24	getuid		
25	stime		
26	ptrace		
27	alarm		
28	fstat		
29	pause		

Example:

Write a line to the terminal using REQUEST.

```
LIST
10 A$="Printed with REQUEST"+CHR$(10)
20 REM Request code for Write is = 4
30 REM Par1 is file descriptor for standard output = 1
40 REM Par2 is pointer to string to write
50 REM Par3 is length of string to write
60 R=REQUEST(4,1,VARPTR(A$),LEN(A$))
*basic*
RUN
Printed with REQUEST
*basic*
```

STATUS (Command)

format:
STAT[US]

This command gives status information about the basic interpreter:
Modes: FLOAT/INTEGER, SINGLE/DOUBLE, NO EXTEND/EXTEND, LONG INT/SHORT INT.
Program size, data size.

INP (changed)

format:
INP(fnr,addr)

The INP function has been changed to work through a special driver accessed as a file. The driver should be opened with an ordinary OPEN statement.
The address should be put together as the card select * 256 plus the port number.
The INP function is executed as "lseek" on specified file and then "read" of one byte.

Example:

```
10 OPEN "/dev/DBinoutb" AS FILE 1
20 I=INP(1,Card*256+Port)
```

OUT (changed)

format:

```
OUT #fnr address,val[address,val...]
```

The OUT statement corresponds to the INP function and also works through a channel.
The address should be put together as the card select * 256 plus the port number.
The OUT statement is executed as a "lseek" on specified file to 'port'. All consecutive 'val' with same 'address' are then written with "write" in one request. If a new 'address' is found a new "lseek" is executed.

GRAPHIC STATEMENTS (ABC1600 or ABC806 with graphic prom)

For all the statements the 'colnr' and 'pattern' parameters are optional, if not specified the last used values are maintained.
For monochrome terminal (ABC1600) 'colnr'=1 and 'pattern'=0 are used to draw comlete lines, arcs ...
'colnr'=0 and 'pattern'=0 are used to clear.

FGFILL x,y[,colnr][,pattern]

Fill rectangular area from previous graphic cursor position to 'x','y'.

Ex.

```
FGFILL 100,100,1,1
```

FGLINE x,y[,colnr][,pattern]

Draw line from previous graphic cursor position to 'x','y'.

Ex.

```
FGLINE 100,100,1,1
```

FGPOINT x,y[,colnr][,op]

Move graphic cursor to 'x','y' and alter pixel according to 'op': 0 set pixel
     1 clear pixel
     2 complement pixel

Ex.

```
FGPOINT 100,100,1,2
```

FGPAINT x,y [,colnr] [,pattern]

Start paint from 'x','y'. After operation graphic cursor is left in 'x','y'.
If 'pattern' is 0 a complete "go around the

corner" paint is done otherwise it just starts
at 'x','y' and goes outwards.

Ex.
```
    FGPAINT 100,100,1,1
```

FGCSEG x,y,len [,colnr] [,pattern]

Draws circle segment from graphic cursor position counter clockwise with origo in 'x','y'.
The "length" of the segment is specified with 'len' in number of vertical and horizontal pixel steps,
This means that a full circle is generated with len= 8 * radius.

Ex.
```
    FGCSEG 100,100,300,1,1
```

MIMER DATABASE OPTION

The MIMER database handler is a relational database management system for creating and maintaining the data base.

The data handled by a relational database system is organized in tables. Every table contains a number of rows, each row consisting of a number of columns.

The rows within a table are maintained in a sorted order. The sorting is done after the primary key (or primary index) defined for each table.

A primary key is one column or several consecutive columns. The primary key column(s) must be defined in the beginning of the row. Two rows in the same table may not have identical primary keys.

All row values in a specific column are of the same type (character, integer or float) and the same length.

Example:

Table CAR is used in the examples below, it consists of four columns all of character type.

Column name	Size in bytes	Note
REGNR	6	Primary key
MODEL	15	
COLOR	8	
YEAR	2	

Table CAR

REGNR	MODEL	COLOR	YEAR
ABC123	GOLF	BLACK	82
BAR762	VOLVO	WHITE	83
HIK093	SAAB	GRAY	79

MIMER statements

The MIMER option contains a number of statements for advanced handling of Mimer tables.

Statement	Abbreviation	Description
MIMER BEGIN	MIMBE	Start Mimer session
MIMER OPEN	MIMOP	Open databank and table
MIMER GETFIRST	MIMGF	Read first row in table
MIMER GETNEXT	MIMGN	Read next row in table
MIMER WRITE	MIMWR	Insert a new row in table
MIMER UPDATE	MIMUP	Update current row in table
MIMER DELETE	MIMDE	Delete current row in table
MIMER TRANSACTION	MIMTR	Start transaction handling
MIMER COMMIT	MIMCO	Make changes permanent
MIMER ABORT	MIMAB	Do not make the changes
MIMER END	MIMEN	Terminate the Mimer session

MIMER BEGIN

Function: Starts the mimer session by establishing contact with the Mimer database handler.

Mode: Direct/Program

Format: MIMER BEGIN <string1>,<string2>

The abbreviated form 'MIMBE' may be used for 'MIMER BEGIN'.

Arguments:

String1 is the user name.

String2 is the password associated with user name in Mimer.

User name and password should be given as strings or string variables. The user name can be given in lower or upper case characters. If it is in lower case it will be transformed internally to upper case before calling the Mimer handler.

Use: This statement is used as a login procedure to the Mimer database handler. The user name and password is checked by the handler and the access rights for the user determined. The Mimer database handler must be running. Until a MIMER END is executed databanks and tables, accessable for the user, can be handled.

Examples:

Ex. Start session as user "USER1" with password in string variable Passwd$

```
MIMER BEGIN "USER1",Passwd$
```

MIMER OPEN

Function:	Opens Mimer databank and table. Connects the specified Mimer columns with the corresponding variables in a MIMER GETFIRST, MIMER GETNEXT, MIMER WRITE or MIMER UPDATE statement.
Mode:	Direct/Program
Format:	MIMER OPEN "<databank>.<table>" AS FILE nr [ACCESS ac] column1, column2, ...

The abbreviated form 'MIMOP' may be used for 'MIMER OPEN'.

Arguments:

Databank is Mimer databank name, max 8 characters long.

Table is Mimer table name, max 8 characters long.

The ACCESS specification is optional and determines the protection for the databank and the table.

The ACCESS values are:

ac		databank	table
RR		Read	Read
SR		Shared	Read
SS		Shared	Shared
XX	(default)	Exclusive	Exclusive

Column1, ... are the Mimer column names, from which data is to be retrieved in a subsequent MIMER GETFIRST or MIMER GETNEXT statement. The first column name given specifies the column from which data will be read into the first basic variable given in MIMER GETFIRST or MIMER GETNEXT. The second column name refers to the second basic variable, and so on.

The same correspondance is valid for MIMER WRITE and MIMER UPDATE but the data transfer is in the opposite direction, from the basic variable to the Mimer column.

If a column name is specified as an empty string (or empty string variable) no connection to a mimer column will be made, the corresponding basic variable in MIMER GETFIRST or MIMER GETNEXT will not receive any data.

Use: The specified table within databank is opened. If the databank is not already open (from opening another table within the same databank), it is also opened.

The databank and table names are transformed to upper case if needed.

The given column names are searched for in the system table "*TABDEF " and the needed information to handle the table is retrieved. Before the search the column names are transformed to upper case.

If a given column name is not found in the "*TABDEF" table mimer error 111 is generated. This is not a fatal error, the table is opened for subsequent handling but no data transfers will be done for the nonexisting column(s).

Note: If several tables are opened within the same databank the access mode used for the databank when opening the first table will be maintained when opening the next one. This means that the same access mode for databank should be used when opening several tables within the same databank.

Examples:

Ex. 1

Open databank "DB1   " and table "CAR   " with default ACCESS (both databank and table Shared). Check that the column names for table CAR are REGNR, MODEL, COLOR and YEAR.

```
Regnr$="regnr"
MIMER OPEN "DB1.car" AS FILE  7 Regnr$,"model","color",
"year"
```

As shown the column names can be given as string variables.

Ex. 2

```
Regnr$=""
MIMER OPEN "DB1.car" AS FILE 7 Regnr$,"model","year",
"color"
```

In this case the column 'regnr' is not connected and the columns 'year' and 'color' are read in a different order.

Ex. 3

```
Regnr$=""
MIMER OPEN "DB1.car" AS FILE 7 ACCESS RR
Regnr$,"model","year","color"
```

Is equivalent to the previous example except that databank and table is opened for Read only.

CLOSE

Function:	Close an open Mimer table.
Mode:	Direct/Program
Format:	CLOSE [channel no., ...]
Arguments:	Channel no. has the same value as in the MIMER OPEN statement.
Use:	Close the table associated with channel 'channel no.'.
	If no channel number is given all files are closed and an automatic MIMER END is performed.
Note:	If channel number is specified the databank is left open until a MIMER END statement is executed. The reason is that more than one table might be open within the same databank.
Example:	
	Ex. Close table CAR from previous example.
	CLOSE 7

MIMER GETFIRST

Function:	Reads the first row in a Mimer table.
Mode:	Direct/Program
Format:	MIMER GETFIRST #nr, column1 REL restr1 LO column2 REL ..., var1 [, var2, var3 ...]
	The abbreviated form 'MIMGF' may be used for 'MIMER GETFIRST'.
Arguments:	Reads a row of data from the Mimer table associated with 'nr'. The value of the first Mimer column specified in the MIMER OPEN statement is placed in 'var1' and so on. If select conditions are specified they should consist of pairs of a column name and a restriction value with a relational operator REL inbetween.

REL can be one of the following

BW	Begin With	(string)
NB	Not Begin with	(string)
CO	COntains	(string)
NC	Not Contains	(string)
EQ	EQual to	
GE	Greater Equal	
GT	Greater Than	
LT	Less Than	
LE	Less Equal	
NE	Not Equal	

If several pairs are given they should be connected with a logical operator LO. The logical operator LO can be

MAND (Mimer AND) or
MOR (Mimer OR).

Use: Reads the first row of data that satisfies the select conditions, from the Mimer table.

Note: The logical operator MOR has higher precedence than MAND (reversed precedence compared to the normal AND and OR) and parentheses can NOT be used to change this precedence.

Example:

Ex. 1

Search for a car in table CAR with the letters AB in the registration number and with gray color.

```
MIMER OPEN "DB1.car" AS FILE  7 Regnr$,"model","color",
"year"
MIMER GETFIRST #7, "regnr" CO "AB" MAND "color" EQ "gray",
Regnr$,Mod$,Col$,Year$
```

Ex. 2

If the first entry in table CAR is to be read, without any select condition, the select field is left empty. The select field is positioned between the first to commas in the MIMER GETFIRST statement.

```
MIMER GETFIRST #7 ,,Regnr$,Mod$,Col$,Year$
```

Ex. 3

When all columns in the table are of string type (in Mimer this corresponds to 'C' type) all columns can be read into one Basic variable.

```
MIMER GETFIRST #7 ,,datarec$
```

Ex. 4

It is not necessary to read all columns in the table just skip the Basic variable name corresponding to that column.

```
MIMER GETFIRST #7 ,,Regnr$,,Col$,Year$
```

Ex. 5

If the Basic variable and the corresponding Mimer column are of different types, type conversion is attempted.

The YEAR column in table CAR is of string type but contains an integer value. Read the year to an integer variable in Basic.

```
MIMER GETFIRST #7 ,,Regnr$,Model$,Col$,Year%
```

MIMER GETNEXT

Function:	Reads the next row in the Mimer table.
Mode:	Direct/Program
Format:	MIMER GETNEXT #fnr , var1 [, var2 ...] The abbreviated form 'MIMGN' may be used for 'MIMER GETNEXT'.
Arguments:	Var1, var2 ... are the Basic variables were the data from the Mimer table columns should be put.
Use:	After a MIMER GETFIRST this statement reads subsequent rows in the table. The different ways of specifying Basic variables mentioned in the description of MIMER GETFIRST are possible in MIMER GETNEXT as well. The main difference is that there are no select condition field. The different MIMER GETNEXT statements can use different Basic variables for data retrieval.

Example:

Ex. 1

```
MIMER GETNEXT #7 ,Regnr1$,Model1$,Col1$,Year1$
MIMER GETNEXT #7 ,Regnr2$,Model2$,Col2$,Year2$
```

MIMER WRITE

Function: Insert a new row in the Mimer table.

Mode: Direct/Program

Format: MIMER WRITE #fnr , var1, var2 ...

The abbreviated form 'MIMWR' may be used for 'MIMER WRITE'.

Arguments: Var1, var2 ... are the Basic variables containing the data to enter in the columns in the Mimer table.

Use: Inserts a new row in the Mimer table. If one or more of the columns are not specified the corresponding mimer columns will be filled with spaces.

Note: If the primary index is the same for the new entry and an entry already in the table, the insertion is not performed.

Var1, var2 and so on must be variables, string constants or numerical constants are not allowed.

Example:

Ex. 1

Enter a new row in table CAR

```
Regnr$="ACC123"
Model$="VOLVO 760"
Col$="GRAY"
Year$="84"
MIMER WRITE #7 ,Regnr$,Model$,Col$,Year$
```

MIMER UPDATE

Function: Change the current row in a Mimer table.

Mode: Direct/Program

Format: MIMER UPDATE #fnr , var1, var2 ...

The abbreviated form 'MIMUP' may be used for 'MIMER UPDATE'.

Arguments: Var1, var2 are the Basic variables containing the new values to be entered in the Mimer table.

Use: After a successful MIMER GETFIRST or MIMER GETNEXT to find the row that should be updated, change the columns that should be updated and do a MIMER UPDATE.

Note: A successful MIMER GETFIRST or MIMER GETNEXT must have been executed before using MIMER UPDATE.

Example:

Ex. 1

Find the car with registration number ABC123 and change the color to BLACK.

```
MIMER GETFIRST #7 ,"regnr" EQ "ABC123",Regnr$,Model$,
Col$,Year$

Col$="BLACK"
MIMER UPDATE #7 ,Regnr$,Model$,Col$,Year$
```

MIMER DELETE

Function: Delete a row in a Mimer table.

Mode: Direct/Program

Format: MIMER DELETE #fnr

The abbreviated form 'MIMDE' may be used for 'MIMER DELETE'.

Argument: Fnr is the channel number associated with a file opened with MIMER OPEN.

Use: Find the row that is to be deleted with a MIMER GETFIRST or MIMER GETNEXT. Delete the current row by executing a MIMER DELETE statement.

Note: A successful MIMER GETFIRST or MIMER GETNEXT must have been executed before using MIMER DELETE.

Example:

Ex. 1

Find the car with registration number ABC123 and delete the row.

```
MIMER GETFIRST #7 ,"regnr" EQ "ABC123",Regnr$,
Model$,Col$,Year$
MIMER DELETE #7
```

MIMER TRANSACTION

Function: Initiates transaction handling.

Mode: Direct/Program

Format: MIMER TRANSACTION #fnr

The abbreviated form 'MIMTR' may be used for 'MIMER TRANSACTION'.

Argument: Fnr is the channel number associated with a file opened with MIMER OPEN.

Use: All subsequent MIMER WRITE, MIMER UPDATE and MIMER DELETE operations are put on an intention list and the changes are only entered into the tables when MIMER COMMIT is executed. If MIMER ABORT is executed, none of the changes done since last MIMER TRANSACTION are moved into database.

Note: The transaction handling works on databank level.

Example:

See MIMER ABORT example below.

MIMER COMMIT

Function: All Mimer operations performed on the databank since MIMER TRANSACTION are made permanent.

Mode: Direct/Program

Format: MIMER COMMIT #fnr

The abbreviated form 'MIMCO' may be used for 'MIMER COMMIT'.

Argument: Fnr is the channel number associated with a file opened with MIMER OPEN.

Use: All MIMER WRITE, MIMER UPDATE and MIMER DELETE operations on the intention list are entered into the tables when MIMER COMMIT is executed.

Note: The transaction handling works on databank level.

Example:

See MIMER ABORT example below.

MIMER ABORT

Function: Transactions on intention list are rolled back.

Mode: Direct/Program

Format: MIMER ABORT #fnr

The abbreviated form 'MIMAB' may be used for 'MIMER ABORT'.

Argument:

Fnr is the channel number associated with a file opened with MIMER OPEN.

Use:

All MIMER WRITE, MIMER UPDATE and MIMER DELETE operations on the intention list are rolled back. No changes since the previous MIMER TRANSACTION are done in the tables when MIMER ABORT is executed.

Note:

The transaction handling works on databank level.

Example:

Ex. 1

Transaction handling is used when one or several tables in the same databank need updating and it is essential that either all updates or none are made.

Assume table WAGE contains one column MONTH and one column SALARY and table EMPLOYEE contains columns EMPNO (employee number) and RECSAL (received salary). In June employee with EMPNO 123 should receive 1000 dollars, this amount should be subtracted from the SALARY value for June in table WAGE and at the same time added to RECSAL value in table EMPLOYEE for the employee concerned.

```
 10 MIMER OPEN "dbl.wage" AS FILE 1 "month","salary"
 20 MIMER OPEN "dbl.employee" AS FILE 2 "empno","recsal"
 30 MIMER GETFIRST #1,"month" EQ "June",Month$,Salary%
 40 MIMER GETFIRST #2,"empno" EQ 123,Empno%,Recsal%
 50 Salary%=Salary%-1000 ! Salary paid is negative
 60 Recsal%=Recsal%+1000 ! Received salary is positive
 65 REM Use transaction handling, NOTE file nr 1 or 2
 66 REM can be used.
 70 MIMER TRANSACTION #1 ! Use transaction handling
 80 ON ERROR GOTO 130
 90 MIMER UPDATE #1,Month$,Salary%
100 MIMER UPDATE #2,Empno%,Recsal%
110 MIMER COMMIT #1 ! Make the changes permanent
120 END
130 MIMER ABORT #1 ! Something went wrong
140 PRINT "Could not update"
150 ON ERROR GOTO
160 STOP
```

MIMER END

Function: Terminates a mimer session, closing all tables and databanks.

Mode: Direct/Program

Format: MIMER END

The abbreviated form 'MIMEN' may be used for 'MIMER END'.

Use: Terminates the mimer session. Closes all open tables and databanks.

Example:

```
10 MIMER BEGIN "USER1",Passwd$
20 MIMER OPEN "DB1.car" AS FILE 7
30 MIMER GETFIRST #7 ,,Regnr$,Model$,Color$,Year$
40 PRINT Regnr$;Model$;Color$;Year$
50 CLOSE 7
60 MIMER END
```

9.3 Mimer error handling

Mimer errors detected by basic are:

100 Relational operator expected
BW, NB ... is missing

101 Logical operator expected
MAND or MOR missing

103 Access mode expected
RR, SR, SS or XX missing

110 To many columns
At present a maximum of 10 columns
in the mimer table can be handled by basic

111 Column name mismatch
The column names given at MIMER OPEN are
not the same as the column names read in
mimer system table *TABDEF

112 Wrong number of columns
The number of variables given for storage
are to few or to many compared to the
number of columns in the mimer table

113 Not a mimer file
The file number specified does not refer
to a file opened with MIMER OPEN

114 Projection problem
Attempt to write or update mimer table
from a variable that does not contain enough
data
or
mimer column of integer or float type can
not be stored in string variable (alignment
restrictions)

115 To many mimer files
At present a maximum of 4 mimer tables
can be opened simultaneously

116 Entry exists
Attempt to insert a new row with a primary index
identical to already existing row.

117 Mimer begin error
Wrong user name or password.

118 Not logged in
No successful MIMER BEGIN statement has been executed.

All errors returned from the Mimer handler are detected and displayed as Basic errors.

The codes consists of four digits the first two specifying the routine in which error occured, and the last two the error type.

First two digits:

11xx	BEGIN2
12xx	OPEND2
13xx	BEGTR2
14xx	ENDTR2
15xx	CLOSD2
16xx	END2
21xx	OPENT2
22xx	PROJE2
23xx	SELEC2
24xx	SET2
25xx	PUSH2
26xx	POP2
27xx	CLOST2
28xx	DEQUE2
31xx	GET2
32xx	INSER2
33xx	UPDAT2
34xx	DELET2
35xx	LOAD2
36xx	DROP2

Last two digits:

xx11	First argument value incorrect
xx12	Second argument value incorrect
xx13	Third argument value incorrect
xx14	Fourth argument value incorrect
xx15	Fifth argument value incorrect
xx16	Sixth argument value incorrect
xx17	Seventh argument value incorrect
xx18	Eight argument value incorrect
xx21	Requested operation not allowed
xx22	Tried to add OR-condition after SET2
xx23	Tried to grant privilege on TRANSDB or LOGDB
xx24	Databank has already been opened
xx25	Table has already been opened (X access involved)
xx26	Tried to remove last primary key incorrectly
xx27	Tried to exceed maximum number of columns
xx28	Tried to change a non empty table
xx29	Tried to exceed maximum row-length
xx31	System control block area exhausted

xx32	No access rights for requested operation
xx33	Tried to use transaction handling without TRANSDB
xx34	Transaction management required
xx35	Operation not allowed, SYSDB opened for read only
xx41- xx49	Databank open error (installation dependent)
xx51	System databank identifier area exhausted
xx52	Tried to open a non-MIMER databank
xx53	Tried to open a non-restarted databank
xx54	Tried to open a write-protected databank

Application example: Car register

This is a listing of a simple car register application.

```
  10 ! ************************************************************
  20 !
  30 ! Car register
  40 !
  50 ! ************************************************************
  60 EXTEND
  70 DEF FNStart
  80   ; CLS;
  90   ; "                  CAR REGISTER"
 100   ;
 110   ; "Password:"
 120   GET A$
 130   IF A$<>CHR$(10) THEN
 140     Pword$=Pword$+A$
 150     GOTO 120
 160   IFEND
 170   ON ERROR GOTO 250
 180   ; "Starting, wait";
 190   MIMER BEGIN "USER",Pword$
 200   Pword$="                    " ! Erase the password
 210   ; CHR$(13)+"STARTED            "
 220   MIMER OPEN "DB1.car" AS FILE 1 "regnr","model","color","year"
 230   ON ERROR GOTO
 240   RETURN 0
 250   ; : ; "Wrong password, try again"
 260   GOTO 110
 270 FNEND
 280 DEF FNMeny
 290   ; CLS;
 300   ; "                     Car register"
 310   ;
 320   ;
 330   ; "                     1. Search"
 340   ; "                     2. Delete"
 350   ; "                     3. List"
 360   ; "                     4. New registration"
 370   ;
 380   INPUT "Select: " S
 390   IF S=1
 400     R=FNSearch
 410   ELIF S=2
 420     R=FNDel
 430   ELIF S=3
 440      R=FNList
 450    ELIF S=4
 460      R=FNIns
 470    ELSE
 480      MIMER END
 490      RETURN 0
 500    IFEND
 510    ; CUR(23,0) "More(y/n)?";
 520    GET A$
 530    IF A$='y' GOTO 290
 540    MIMER END
 550    RETURN 0
 560  FNEND
 570  DEF FNOut(X,Y)
 580    ; CUR(X,Y) Regnr$,Model$,Color$,Year$
 590    RETURN 0
 600  FNEND
 610  DEF FNSearch
 620    INPUT "Reg nr:",Regnr$
 630    ON ERROR GOTO 690
 640    MIMER GETFIRST #1,"regnr" EQ Regnr$,Regnr$,Model$,Color$,Year$
 650    ; CLS;"Found:"
 660    Entry=1
 670    R=FNOut(2,5)
 680    RETURN 0
 690    ; CLS;"NOT FOUND"
 700    ON ERROR GOTO
 710    RETURN 0
 720  FNEND
 730  DEF FNList
 740    ; CLS;"                        Car register"
 750    MIMER GETFIRST #1,,Regnr$,Model$,Color$,Year$
 760    I=1
 770    ON ERROR GOTO 820
 780    R=FNOut(I,0)
 790    I=I+1
 800    MIMER GETNEXT #1,Regnr$,Model$,Color$,Year$
 810    GOTO 780
 820    ON ERROR GOTO
 830    ; "Number of registered";I-1
 840    Entry=0
 850    RETURN 0
 860  FNEND
 870  DEF FNDel
 880    IF Entry=0 THEN ; "No current row, search first!" : RETURN 0
 890    ON ERROR GOTO 930
 900    MIMER DELETE #1
 910    ; "Deleting:";Regnr$,Model$,Color$,Year$
 920    RETURN 0
 930    ; "Can't delete"
 940    ON ERROR GOTO
 950    RETURN 0
 960  FNEND
 970  DEF FNIns
 980    ; CLS;"New registration"
 990    INPUT "Reg number:",Regnr$
1000    INPUT "Model:",Model$
1010    INPUT "Color:",Color$
1020    INPUT "Model year:",Year$
1030    ON ERROR GOTO 1070
```

```
1040   MIMER WRITE #1,Regnr$,Model$,Color$,Year$
1050   ON ERROR GOTO
1060   RETURN 0
1070   ; "Can't make insertion"
1080   RETURN 0
1090 FNEND
1100 ! ***********************************
1110 !
1120 ! main program
1130 ! ***********************************
1140 R=FNStart
1150 R=FNMeny
```

FILES

```
 /bin/basic  standard basic
 /bin/mdbasic  mimer basic
 /usr/etc/bascap  editor input description
 /usr/etc/basicerr.txt error messages
 /usr/etc/sortorder.tab ISAM ascii sort order desc.
 /usr/etc/translate.txt OPEN statement translation
 /etc/termcap  terminal capabilities
```