DAInamic Software & Library
c/o W. Hermans
Mottaart 20 B-3170 HERSELT

D - BASIC

DAInamic software

D - BASIC

WELCOME TO THE WORLD OF DBASIC...

Soon after he got his DAIpc, Willy Coremans joined DAInamic redaction and told us he needed a lot of extra commands and features for easy programming. We all agreed, but we did not believe this was possible without a lot of CALLM's,mlp-trics and POKING around in system ram.

Here it is now : a complete integrated BASIC-extension, bringing DAI-BASIC close to PASCAL and other structured programming languages.

We want to thank Willy Coremans for his one-year super-job, Frans Couwberghs did the typing work of this manual, and wdw did the final revision.

We also want to thank you if you send your suggestions, ideas, appreciations about DBASIC to :

Willy Coremans
Hoekheide 27
B 3140 RAMSEL

tel 016/697419

W.Hermans

INTRODUCTION

DBASIC V2.1 is a very powerfull high-level language written for the DAI-personal computer.
Besides, DBASIC V2.1 is 100 % compatible whith the ROM-resident BASIC V1.0 and BASIC 1.1 of the DAI-pc.
DBASIC V2.1 is a software product, no hardware adaptions are needed.
DBASIC V2.1 adds to the existing set of BASIC commands and statements a number of new commands and statements.
These new statements are principally added for structured programming.
Special graphical commands (ex. Turtle-graphics) or commands to operate I/O-devices (ex. Memocom-MDCR) are not included on purpose : Not everyone will use the graphic possibilities of the DAI-pc or will have a DCR-drive connected, therefore some code would be useless and decreases the amount of memory available for DBASIC programs.
However, if neccessary, these special graphical commands or I/O-driving packages can be integrated in DBASIC.They can be grouped in so-called DBASIC-EXTENSIONS, loaded and relocated or deleted when needed.
For explanation on these extensions, see the appropriate documentation. (Appendix D and E)

How to read this manual

This manual is divided into four chapters.

Chapter 1 : Covers a variety of topics.
Chapter 2 : Describes the DBASIC-statements.
Chapter 3 : Describes the added intrinsic functions.
Chapter 4 : Handles about procedures and functions.

Some helpfull information is gathered in appendix A to E

```
! Please note that only information is given on !
! substantially new topics. For syntax description !
! of already existing statements and functions !
! refer to the DAI-pc manual or (for Dutch-speaking !
! people) to 'Gestructureerd programmeren met !
! DAI-BASIC' from Bruno Van Rompay.              !
```

CHAPTER 1

1 General information about DBASIC V2.1

1.1 INITIALISATION.

At this moment two versions of DBASIC V2.1 exist, a Memocom-MDCR version and an audio-cassette version.
The DCR-version starts whith a USER-file, so resetting the DAI-pc while the cassette without write-plug is in DCR-drive Ø will automatically start DBASIC V2.1. Besides DBASIC V2.1, the Memocom MDCR-driving package is automatically loaded and relocated. (See Appendix E)

The audio-cassette version starts whith an autostart loader. (Type : *UT, >Z3, >R)

1.2 OPERATION-MODES

Just like every BASIC-version, DBASIC can be used in two different modes of operation.
DBASIC can be used in the direct-command mode and in program-mode.
In the direct-command mode a limited set of commands can be used.They are executed immediatly.
DBASIC is in the program-mode during execution of a program stored in memory.
Before ececution, in both modes, the commands and statements will be compiled first.
During compilation the program and/or command line is checked for structural errors.
Compilation is, either executed automaticaly, (after typing the command line or, for a program, after giving the RUN command) or after giving the COMPILE command.(See chapter 2)

1.3 PROGRAM FORMAT

A DBASIC program consists of a number of program lines.
A program line has the following format :
(Square brackets indicate optional parts)

nnnnn [<label>] statement [:statement...] <carriage return>

1.3.1 LABELS

Program lines can be identified by a label. Statements refering to a line by label have been build in.

Format : <label> : "<name>
<name> : Up to 14 alpha-numeric characters,first character must be alphabetic.

Remarks : All commands and statements, except LIST and EDIT which refer to a line by linenumber can also refer to that line by a label.

1.4 ARRAYS

A limitation of array-handling power in BASIC V1.Ø and V1.1 is the maximum of 255 elements/dimension.
DBASIC specifies a maximum dimension that can be changed by the user. (Default = 2000)

To change the maximum dimension your program should include for instance :

```
10 DIMMAX%=5000
20 DOKE #56,DIMMAX%
```

These two statements will set the maximum dimension to 5000.

1.5. EXPRESSIONS

Most DBASIC statements accept one or more expressions for their arguments. Besides 'intrinsic' functions, variables and constants, an expression can contain also 'user defined' functions.
These functions can be defined by DEF FN or FUNCTION. (See chapter 2)

NOTE : -No assumption of the expression type is made : defined functions can be used in logical expressions as well as in mathematical- or string- expressions.
-The priority of 'user defined' functions is the same as the priority of 'intrinsic' functions.

1.6 ERROR TRAPPING

In DBASIC all runtime errors can be trapped.
Compile time errors, either during semi-compilation from the edit-buffer, from screen or during compilation whith COMPILE or RUN commands, cannot be trapped.
For a detailed analysis of the error which occured, the ERR, ERL and ERR$ 'intrinsic' functions can be used. (See chapter 3 and appendix B and C)
A complete list of error-messages is given in appendix A.

1.7 COMPILATION

Before starting program or command-line execution, the program or the command line has to be compiled first.
Compilation is done in DBASIC pseudo-code.
Compilation will be done automatically after typing a command line (Compilation of the command-line) and after giving the RUN command. (Compilation of the program)
Before saving a program, you can compile it by the COMPILE command, so when you LOAD the program, it automatically start execution without recompilation first.
(See chapter 2)

1.8 Protection.

To gain control over the ROM-resident operating system, system-RAM locations 0H to 40H had to be changed drasticaly. To protect the user against changing this system-RAM, no access of memory-locations 0H to 40H is allowed anymore. Trying to access these memory-locations will result in a 'NUMBER OUT OF RANGE' error. Also in utilyties the access of these addresses is forbidden.

1.9 Implicit integer.

When loading DBASIC the default type for constants and for all variables will be set to implicit integer.
In all examples of this manual, variables will be considered as implicit integer unless otherwise stated.

CHAPTER 2

DBASIC COMMANDS AND STATEMENTS

Only substantially new DBASIC commands and statements are described in this chapter. Each description is formatted as follows :

Format : Shows the correct format for the instruction.
See below for format notation.
Valid : Tells wether it can be used as command only, in program only or both.
Purpose : Tells where the instruction is used for.
Remarks : Describes in detail how the instruction is used.
Example : Shows sample programs or program segments that demonstrate the use of the instruction.

Format notation

Wherever the format of a statement is given, the following rules apply :

1. Items in capital letters must be input as shown.
2. Items in lower case letters enclosed in angle brackets (<>) are to be supplied by the user.
3. Items in square brackets are optional.
4. All punctuation except angle brackets and square brackets (i.e., comma's, parentheses, semicolons, hyphens, equal signs) must be included where shown.
5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).

2.1 BREAK

Format : BREAK ON
BREAK OFF

Valid : statement

Purpose : To enable or disable interruption of the program.

Remarks : After executing a BREAK OFF statement, program execution cannot be suspended anymore by typing the BREAK-key.
However the BREAK-key will not be disabled completely.
Stopping your cassette-drive or the sound-generator is still possible.

Example :
```
10  BREAK OFF:REM NO INTERRUPTION ALLOWED
...
100 BREAK ON: REM ALLOW INTERRUPTION
```

2.2 COMPILE

Format : COMPILE

Valid : direct command

Purpose : To prepare a program for execution and to check on structural errors.

Remarks : The compiler will be called automaticaly after :
- typing a command line in direct-command mode to compile this command line before execution.
- giving the RUN command if the program in memory has not been compiled yet.
  Thus when loading an already compiled program compilation will be skipped.

A compiled program will be autostart.
A list of all errors dedectable during compilation is given in appendix B.
Labels, procedures and functions will remain undefined untill the program has been compiled.

2.3 CONTINUE

Format : CONTINUE

Valid : command/statement

Purpose : To continue program execution after the BREAK-key has been typed or a STOP statement has been executed.

Remarks : As direct-command CONTINUE executes as the CONT command.
However, in a program, CONTINUE will end the interrupt service routine (started by typing the BREAK-key) and resumes program execution at the point where it was interrupted.

Example :
```
10    ON BREAK GOTO "INTERRUPT
...
...
65000 "INTERRUPT CURSOR 0,0
65010 PRINT "NO BREAK ALLOWED";:WAIT TIME 20
65020 CURSOR 0,0:PRINT SPC(60);
65030 CONTINUE
```


2.4 DEF FN / FUNCTION

Formats : DEF FN <function name> (<parameter list>):
<function definition>:END FN

or : DEF FN <function name> (<parameter list>)=
<expression>

with :
- <function name> : <name>[<type indicator>]
- <type indicator> : !, % or $
- <parameter list> : <value-par> VAR<var-par> ARR<arr-par> FN<fn-par>
- <value-par> : subscripted or unsubscripted variable-references separated by commas.
- <var-par> : unsubscripted variable names.
- <arr-par> : subscripted variable names.
- <fn-par> : unsubscripted variable names
- <function definition> : DBASIC statements containing at least one FN = statement.

note : The DEF FN keyword can be replaced by the FUNCTION keyword.

Valid : statement

Purpose : To define a function that is written by the user.

Remarks : The value-parameters will be assigned a value equal to the evalution of the matching expression in the caller.

The variable-parameters will refer to the matching unsubsripted variables in the function caller.
These variable-parameters can be used to transfer values, condition-codes, messages etc... to variables not used in the function.

The array-parameters will refer to the matching subscripted variables in the function caller, denoted as <name>[<type indicator>]().
Array-parameters can be used to transfer complete arrays between the function-caller and the function.

Function-parameters will refer to the matching expressions in the function-caller.
Thus functions can perform different calculations, string-handling etc... using different expressions in the function-caller.

Example : Faculty of a number.

```
10   FUNCTION FAC(I)
20   IF I=0 THEN FN=1
30   ELSE FN=I*FAC(I-1):END IF
40   END FN
...
```

Note : For more explanation and more examples see CHAPTER 4 : Procedures and functions.

2.5 DEF PROC / PROCEDURE

Format : DEF PROC <name> [<parameter list>]:
<procedure definition>:END PROC

with : parameter list>: see DEF FN / FUNCTION
<procedure definition> :DBASIC statements.

note : The DEF PROC keyword can be replaced by the PROCEDURE keyword.

Valid : statement.

Purpose : To define a procedure written by the user.

Remarks : The procedure parameter-list obeys exactly the same rules as the function parameter-list.

Example : Simulate HOME (Apple-II)

```
10  DEF PROC HOME:PRINT CHR$(12);:END PROC
...
100 "START HOME
110 CLEAR 2000:...
```

Note : For explanation and more examples see CHAPTER 4 : PROCEDURES AND FUNCTIONS

2.6 LOCAL

Format : LOCAL <variable 1>,...,<variable i>,...,<variable n>

Valid : statement

Purpose : To define local variables in a procedure or function.

Remarks : The variables <variable 1) to <variable n> specified in the LOCAL statement will be used local in the function or procedure : i.e the value assigned to this variables before calling the the procedure or function will not have been changed after the procedure or function call.

Example :

```
10  PROCEDURE SWAP VAR X$,Y$
20 LOCAL HULP$
30  HULP$=X$:X$=Y$:Y$=HULP$
40  END PROC
100 A$=HELLO :B$="BYE"
110 HULP$="THIS VARIABLE WILL NOT BE CHANGED"
120 SWAP A$,B$
130 PRINT A$,B$,HULP$
...
```


2.7 DOKE

Format : DOKE <integer expression 1>,<integer expression 2>

Valid : statement/command

Purpose : To put the 2-byte value <integer expression 2> at address location <integer expression 1>, low byte first.

Remarks : This statement is analog to the already existing POKE statement.

Example :

```
10 FOR I=#3000 TO #4000 STEP 2
20 DOKE I,0
```

2.8 ERROR

Format : ERROR <integer expression>

Valid : statement/command

Purpose : To simulate the occurance of a DBASIC error, or to allow errorcodes to be defined by the user.

Remarks : The value of <integer expression> must be in the range [0,255].
If the value of <integer expression> equals an error code already in use by DBASIC, the error statement will simulate the occurance of that error and the corresponding error-message will be printed.

To define your own error code, use a value that is larger than any used by DBASIC. (55 in DBASIC V2.1) This user-defined error code may then be conveniently handled in an error trap routine.

See also appendix C : Error reporting.

Example :

```
10  ON ERROR GOTO "TRAP
...
100 CURSOR 0,10
110 INPUT "PASSWORD : ";P$
120 IF P$<>PASSWORD$ THEN ERROR 200:END IF
...
10000 "TRAP IF ERR=200 THEN CURSOR 0,0
10010 PRINT "UNAUTHORISED USER";:RESUME 100
10020 ELSE RESUME NEXT:END IF
```

2.9 GOSUB <label>

Format : GOSUB <label>

Valid : statement

Purpose : To call a subroutine which starts at <label>.

Remarks : A subroutine must contain at least one RETURN statement to branch back to the statement following the most recent GOSUB statement.

A subroutine may be called any number of times in a program, and may be called from within another subroutine. Such nesting of subroutines is limited by the available stack-memory.

Example :
```
10   PRINT CHR$(12);"FOR EXPLANATION TYPE 'H'"
20   REPEAT CHAR=GETC:UNTIL CHAR<>0
30   IF CHAR=ASC("H") THEN GOSUB "HELP:END IF
...
1000 "HELP PRINT CHR$(12);"THIS PROGRAM WILL ...
```

2.10 GOTO <label>

Format : GOTO <label>

Valid : statement

Purpose : To proceed program execution at <label>.

Example :
```
10    GOTO "INIT
20    "START HOME:...
...
65000 "INIT CLEAR 5000:...
...
65100 GOTO "START
```


2.11 IF ... THEN ... ELSE ... END IF

Format : IF <logical expression> THEN [<statement(s)>]
[:ELSE [<statement(s)]]:END IF

Valid : statement

Purpose : If the result of <logical expression> is true, the THEN clause is executed.
If the result of <logical expression> is false, the THEN clause is ignored and the ELSE clause, if present, is executed.
Execution continues with the statement following END IF.
Both the THEN and ELSE clause can be several program lines long.
IF ... THEN ... ELSE ... END IF statements may be nested. Nesting is only limited by the available memory.

Example :
```
...
100 IF A<B THEN PRINT
110 PRINT "A<B"
120 ELSE IF A>B THEN PRINT
130 PRINT "A>B"
140 ELSE PRINT
150 PRINT "A=B"
160 END IF
170 END IF
```

2.12 IF ... GOTO <label>

Format : IF <logical expression> GOTO <label>

Valid : statement

Purpose : To make decision regarding program flow based on the result returned by a logical expression.

Remarks : If the result of <logical expression> is true, program execution will proceed at <label>.
If the result of <logical expression> is false, program execution will proceed with the next statement.

Example :
```
...
1000  PRINT "TO END TYPE 'E'"
1010  REPEAT CHAR=GETC:UNTIL CHAR<>0
1020  IF CHAR=ASC("E") THEN "FIN
...
60000 "FIN END
```

2.13 ON BREAK GOTO

Format : ON BREAK GOTO <linenumber)
ON BREAK GOTO <label>
ON BREAK OFF

Valid : statement

Purpose : To enable interrupt trapping and specifie the first line of the interrupt handling routine.

Remarks : Once interrupt trapping has been enabled, pressing the BREAK-key will cause a jump to the specified interrupt handling routine.
To end the interrupt handling routine a CONTINUE statement should be included in the program.
To disable interrupt trapping a ON BREAK OFF statement should be executed.

Example :
```
...
20    ON BREAK GOTO "INTERRUPT
...
65000 "INTERRUPT PRINT "CURRENT STATE : ";CURSTAT$
65010 CONTINUE
```

2.14 ON ERROR GOTO

Format : ON ERROR GOTO <linenumber>
ON ERROR GOTO <label>
ON ERROR OFF

Valid : statement

Purpose : To enable error trapping and specifie the first line of the error handling routine.

Remarks : Once error trapping has been enabled all errors detected, including direct mode errors, will cause a jump to the specified error handling subroutine. To disable error trapping, execute an ON ERROR OFF statement. Subsequent errors will print an error message and halt execution. An ON ERROR OFF statement that appears in an error trapping subroutine causes DBASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping routines execute an ON ERROR OFF if an error is encountered for which there is no recovery action.

NOTE : If an error occurs during execution of an error handling routine, the DBASIC error message is printed and execution terminates. Error trapping is not done whitin the error handling routine.

Example :
```
10    ON ERROR GOTO 10000
...
10000 IF ERR<>10 THEN ON ERROR OFF
10010 REM --- ONLY TRAP ERROR 10 ---
10020 ELSE ...
```


2.15 ON ... GOSUB <list of labels>

Format : ON <expression> GOSUB <list of labels>

with <list of lables> :

<label 1>,...,<label i>,...,<label n>

Valid : statement

Purpose : To call one of several subroutines beginning at <label>, depending on the value returned by the evaluation of <expression>.

Remarks : The value of <expression> determines which label in <list of labels> will be called.
For example, if the value of <expression> is three the subroutine beginning at the third label in <list of labels> will be called.
When RETURNing from the subroutine execution will proceed with the next statement.

Example :
```
10 "START ON I GOSUB "INIT,"PROCESS,"QUIT
20 I=I+1:GOTO "START
```

2.16 ON ... GOTO <LIST OF LABELS>

Format : ON <expression> GOTO <list of labels>

with <list of labels> :

<label 1>,...,<label i>,...,<label n>

Valid : statement

Purpose : To branch to one of several labels, depending on the value of <expression>.

Remarks : The value of <expression> determines which label in <list of labels> will be branched to.
For example, if the value is two, the second label in <list of labels> will be branched to.

Example :
```
...
100 ON GETC-#40 GOTO "A,"B,"C
```

2.17 RESTORE

Format : RESTORE <linenumber>

RESTORE <label>

RESTORE

Valid : command/statement

Purpose : To allow DATA statements to be reread from a specified line, or from the first DATA statement.

Remarks : If a RESTORE <linenumber> or RESTORE <label> is executed, the next READ statement will read the first item in the first DATA statement following <linenumber> or <label>.

Example :
```
...
100   RESTORE "GRAPHDATA
...
20000 "GRAPHDATA REM --- DATA FOR DRAWING ---
20010 DATA 10,100,0,60,...
...
```

2.18 RESUME

Format : RESUME

RESUME <linenumber>

RESUME <label>

RESUME NEXT

Valid : statement

Purpose : To resume program execution after an error recovery procedure has been performed.

Remarks : Depending upon where execution is to resume,the four formats shown above may be used.

RESUME : Execution resumes at the statement which caused the error.
RESUME <linenumber> : Execution resumes at <linenumber>.
RESUME <label> : Execution resumes at <label>
RESUME NEXT : Execution resumes at the statement immediatly following the one which caused the error.
A RESUME statement that is not in an error trap routine causes a "RESUME WITHOUT ERROR" message to be printed.

Example :
```
10    ON ERROR GOTO "FAULT
...
50000 "FAULT IF ERR=0 THEN HOME
50010 PRINT "RESTART":RESUME 10
50020 END IF:RESUME NEXT
```


2.19 REPEAT ... UNTIL ...

Format : REPEAT [statement(s)]:UNTIL <logical expression>

Valid : command/statement

Purpose : To execute a statement or series of statements in a loop until a given condition is true.

Remarks : The loop statements are executed at least one time.
If <logical expression> is false, they are executed again.
If <logical expression> is true, program execution continues at the statement following the UNTIL statement.

REPEAT/UNTIL loops may be nested with REPEAT/UNTIL FOR/NEXT and WHILE/WEND loops.

Example :
```
...
100 REPEAT A=GETC
110 PRINT A
120 UNTIL A=ASC("S")
...
```

2.20 RUN <label>

Format : RUN <label>

Valid : command

Purpose : To start program execution at <label>.

Remarks : To use this form of the RUN command, the program in memory has to be compiled first.
If the program is not compiled <label> will be undefined and an "UNDEFINED LABEL" error will be generated.

2.21 WHILE ... DO ... WEND

Format : WHILE <logical expression> DO [statement(s)]:WEND

Valid : statement

Purpose : To execute a statement or series of statements in a loop as long as a given condition is true.

Remarks : If <logical expression> is true, the loop statements are executed until the wend statement is encountered. DBASIC then returns to the WHILE statement and checks <logical expression>. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested with WHILE/WEND loops, REPEAT/UNTIL loops and FOR/NEXT loops.

Example : 1.
```
10 PRINT "TYPE ANY KEY TO CONTINUE"
20 WHILE GETC=0 DO WEND:REM WAIT FOR KEY PRESSED.
...
```
2.
```
...
100 RESTORE "STRINGDATA:FLAG=1:I=0
110 WHILE FLAG=1 DO READ A$(I)
120 IF A$(I)="END" THEN FLAG=0
130 ELSE FLAG=1:END IF
140 I=1+1:WEND
...
```
3.
```
...
100 RESTORE "STRINGDATA:I=0:READ A$(I)
110 WHILE A$(I)<>"END" DO I=I+1
120 READ A$(I):WEND
...
```

2.22 FN =

Format : FN =<expression>

Valid : statement

Purpose : To end evaluation of a 'user defined' function and to return <expression> as value.

Remarks : <expression> has to be of the same type as the 'user defined' function where the FN = statement belongs to.
A function definition can contain more than one FN = statement to end evaluation.

Example :
```
10  FUNCTION  LEFT3$(A$)
20   IF LEN(A$)<3 THEN FN =A$
30   ELSE FN=LEFT$(A$,3)
40   END IF
50   END FN
...
100 PRINT LEFT3$("TEST"),LEFT3$("AB")
```


CHAPTER 3

DBASIC 'INTRINSIC' FUNCTIONS.

Only substantially new DBASIC functions are described in this chapter.

Each description is formatted as follows :

Format : Shows the correct format of the function.

Function : Tells what the result of the function will be.

Remarks : Describes in detail how the function is used.

Example : Shows some examples that demonstrate the use of the function.

3.1 DEEK

Format : DEEK(<integer expression>)

Function : Returns the two-byte value at address-location <integer expression>.

Remarks : The low-byte is taken from address-location <integer expression), the high-byte is taken from address-location <integer expression>+1.

Example :
```
10 HEAP=#29B
20 PRINT "THE HEAP STARTS AT ADDRESS #";
30 PRINT HEX$(DEEK(HEAP))
...
```

3.2 DIM

Format : DIM(<array name>,<integer expression>)

Function : Suppose the value of <integer expression> is n, then the function DIM will return the value of the n-th dimension of <array name>.

Remarks : If <integer expression> is less then 1 or <integer expression> is larger then the number of the dimensions of <array name> a 'NUMBER OUT OF RANGE' error is returned.

Example :
```
10 DIM A(10,4)
20 PRINT DIM(A,1),DIM(A,2)
...
```

3.3 ERL

Format : ERL

Function : Returns the linenumber in which an error occured.

Remarks : An error-linenumber equal to zero (0) means it was a direct-command mode error.

Example :
```
10   ON ERROR GOTO "TRAP
20   ERROR 10
...
1000 "TRAP PRINT ERL:RESUME NEXT
```

Note : See also appendix C.

3.4 ERR

Format : ERR

Function : Returns the error number.

Remarks : ERR is in the range [0,255].

Example :
```
10   ON ERROR GOTO "TRAP
20   ERROR 50
...
1000 "TRAP PRINT ERR:RESUME NEXT
```

Note : See also appendix C

3.5 ERR$

Format : ERR$

Function : Returns the identification of the DBASIC-extension which caused the error.

Remarks : If the error was generated by a DBASIC statement or command then ERR$ is empty.

Example : If you are working with the DCR-version of DBASIC V2.1, and the DCR-driving package is on line then typing :

*REW 1000 will give the error message:

DCR FILE NUMBER OUT OF RANGE

*PRINT ERR$ will give :

DCR

Note : See also appendix C, D and E.

3.6 INTEGER

Format : INTEGER(<floating point expression>)

Function : To convert a floating point expression to integer format.

Remarks : There is a significant diffirence between the INTEGER function and the INT function.
The INTEGER function returns an integer value while the INT function returns a floating point value equal to the part of the argument at the left side of the decimal point.

The INTEGER function can be usefull to convert expressions to the integer type before passing them as function parameters to a procedure or user-defined functions.

Example :
```
10  PROCEDURE TEST FN Z
20  PRINT Z
30  END PROC
...
100 FOR I!=0.1 TO 1.0 STEP 0.1
110 TEST INTEGER(SIN(I!)*2.0)
120 NEXT I!
```

3.7 ISTR$

Format : ISTR$(<integer expression>)

Function : Returns a string representing the value of <integer expression>.

Remarks : The ISTR$ function is familiar to the STR$ function.
The ISTR$ function however returns a string without added '.0'.

Example :
```
10 A=100000000
20 PRINT ISTR$(A),STR$(A)
...
```

3.8 IVAR

Format : IVAR(<integer expression>)

Function : Returns the integer value stored at memory-location <integer expression>.

Remarks : The IVAR function can be usefull to point into an integer type array.

Example :
```
5  CLEAR 10000
10 DIM A(10,10,10):LOADA A "INTARR"
20 FOR I=VARPTR(A(0,0,0)) TO VARPTR(A(10,10,10))
   STEP 4
30 PRINT IVAR(I)
40 NEXT I
```

3.9 NDIM

Format : NDIM(<array name>)

Function : Returns the number of dimensions reserved for <array name>.

Remarks : The NDIM function could be used in procedures and functions which contain array-parameters to test the number of dimensions of the arrays passed.

Example :
```
10    REM --- ONE DIMENSION ARRAY SORT ---
20    PROCEDURE SORT ARR A$
30    IF NDIM(A$)<>1 THEN ERROR 100
40    ELSE ...
...
100   "START ON ERROR GOTO "TRAP
...
20000 "TRAP IF ERR=100 THEN PRINT "ONLY ONE
      DIMENSION IS ALLOWED FOR SORTING"
20010 ELSE ...
```

3.10 REAL

Format : REAL(<integer expression>)

Function : To convert an integer expression to floating point.

Remarks : The REAL function can be usefull to convert expressions to the floating point type before passing them as a function parameter to a procedure or a user defined function.

Example :
```
10  PROCEDURE TEST FN Z!
20  PRINT Z!
30  END PROC
...
100 TEST REAL(XMAX)
```


3.11 VAR

Format : VAR(<integer expression>)

Function : Returns the floating point value stored at memory location <integer expression>

Remarks : The VAR function can be usefull to point into a floating point type array.

Example :
```
5  CLEAR 10000
10 DIM A!(10,10,10):LOADA A! "FPTARR"
20 FOR I=VARPTR(A!(10,9,0)) TO VARPTR(A!(10,10,10))
   STEP 4
30 PRINT VAR(I)
40 NEXT
```

3.12 VAR$

Format : VAR$(<integer expression>)

Function : Returns the string to which memory-location <integer expression> points to.

Remarks : The VAR$ function can be usefull to point into a string type array.
It is possible to sort a string type array by just swapping the pointers to the strings.

Example :
```
5  CLEAR 10000
10 DIM A$(100):LOADA A$ "STRARR"
20 FOR I=VARPTR(A$(0)) TO VARPTR (A$(100)) STEP 2
30 PRINT VAR$(I)
40 NEXT
```

CHAPTER 4

PROCEDURES AND FUNCTIONS

In this chapter procedures and functions are explained in more detail . Some examples will be helpfull in understanding.

4.1 DECLARATION.

A procedure or function has to be declared somewhere in the program. Although it is not nessecary, we recommand you to declare them in the begin of the program, as in UCSD PASCAL.

```
Example : 10 DEF PROC HOME
          20 PRINT CHR$(12);
          30 END PROC
```

The procedure HOME, called as HOME, will clear the screen. This very simple procedure could also be declared on one line.

```
Example : 10 PROCEDURE HOME:PRINT CHR$(12);:END PROC
```

Within a procedure or function declaration, other functions or procedures can be declared.

```
Example : 10 PROCEDURE PROMPT
          20 PROCEDURE HOME
          30 PRINT CHR$(12);
          40 END PROC
          50 HOME
          60 PRINT "DBASIC V2.1"
          70 END PROC
```

Procedure PROMPT, called as PROMPT, will clear the screen and print the DBASIC V2.1 prompt in the upper-left corner.
Note that the code executed by procedure PROMPT concists of line 50 and 60, line 20 to 40 is not executed.
In fact line 20 to 40 declare a 'sub-procedure' of the procedure PROMPT.

4.2 VALUE PARAMETERS.

You can pass four types of parameters to a procedure or function. The first and simplest parameter-type is the value parameter.

```
Example : 10 PROCEDURE AST I,J
          20 WHILE I<>0 DO PRINT "*";:I=I-1:WEND
          30 WHILE J<>0 DO PRINT ":";:J=J-1:WEND
          40 END PROC
          50 "START A=4:B=5
          60 AST A+1,B
          70 AST 8,6-3
```

Executing line 60 will print five asterixs followed by five colons, line 70 will print eigth asterixs followed by three colons.

```
Example : 10 FUNCTION FAC!(N)
          20 IF N=0 THEN FN=1.0
          30 ELSE FN=N*FAC!(N-1)
          40 END IF
          50 END FN
          60 FOR I=0 TO 10:PRINT FAC!(N):NEXT
```

Executing line 60 will print 0! to 10!.(faculty)

```
Note that : N!=  1 if N=0
                 N*(N-1)! if N>0
```

Note also that the function FAC! is defined recursively. It can also be defined as an iteration.

```
Example : 10 FUNCTION FAC!(N):LOCAL I,T!:T!=1.0
          20 IF N=0 THEN FN=1.0
          30 ELSE FOR I=1 TO N:T!=T!*I:NEXT
          40 END IF
          50 END FN
```

A number of mathematical functions that are not intrinsic to DBASIC may be calculated as follows :

Function:	DBASIC equivalent:
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN(X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLOC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=EXP(-X)/(EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

Note that all variables are implicit floating point and that X is a value parameter.

4.3 VARIABLE PARAMETERS.

A second type of parameter you can pass to a procedure or function, is the variable parameter.
Variable parameters allow you to transfer unsubscripted variables to and from procedures or functions.

```
Example : 10  PROCEDURE SWAP VAR A$,B$
          20  LOCAL HELP$
          30  HELP$=A$:A$=B$:B$=HELP$
          40  END PROC
          ...
          100 BEGIN$="BEGIN":EN$="END"
          110 SWAP BEGIN$,EN$
          120 PRINT BEGIN$,EN$
```

The effect of the procedure call SWAP BEGIN$,EN$ will be that the contents of variables BEGIN$ and EN$ will have been 'swapped'.
An easy way to understand how the procedure or function will execute is, to substitute the variable parameters by the variables in the caller.
In our example this will give :

```
          ...
          20  LOCAL HELP$
          30  HELP$=BEGIN$:BEGIN$=EN$:EN$=HELP$
          ...
```

```
Example : 10  PROCEDURE TESTSPEED VAR TIM:TIM=0
          20  PRINT "TYPE A KEY AS QUICK AS YOU CAN"
          30  WHILE GETC=0 DO TIM=TIM+1:WEND
          40  END PROC
          ...
          100 TESTSPEED I
          110 PRINT "YOU WAITED";I;" WHILE/WEND LOOPS"
```

In this example Variable I is used only to recieve a value from the procedure TESTSPEED.

4.4 ARRAY PARAMETERS.

Beside unsubscripted variables you can also transfer complete arrays to a procedure or function.

```
Example : 10  DEF FN MAX(ARR A):LOCAL I,MX
          20  FOR I=0 TO DIM(A,1)
          30  IF A(I)>MX THEN MX=A(I)
          40  END IF
          50  NEXT
          60  FN =MX
          70  END FN
          ...
          100 PRINT "THE MAXIMUM PRICE IS";MAX(PRICE())
          101 PRINT "THE  MAXIMUM  TAX IS";MAX(TAX())
          ...
```

In this example PRICE and TAX are one-dimensional integer-type arrays.
Note that the dimension of the arrays can be found with the DIM function.
Note also that the arrays that pass to a procedure or function have to be noted as <array name>().

```
Example : 10  PROCEDURE BSORT ARR A$
          20  DEF PROC SWAPPTR X,Y:LOCAL H
          30  H=DEEK(X):DOKE X,DEEK(Y):DOKE Y,H:END PROC
          40  SFLAG=1
          50  WHILE SFLAG=1 DO SFLAG=0
          60  FOR I=VARPTR(A$(0)) TO VARPTR(DIM(A$,1)-2))
              STEP 2
          70  IF VAR$(I)>VAR$(I+2) THEN SWAPPTR I,I+2:SFLAG=1
          80  END IF
          90  WEND
          100 END PROC
          ...
          1000 BSORT NAME$()
```

This is an example of a sort-routine for one-dimensional string-type arrays.
Note that only the pointers to the strings are reordened.
Note also that procedure BSORT is a very unefficient sort-routine, because in the worst case it scans n times the complete array with n equal to the length of the array.

4.5 FUNCTION PARAMETERS.

A last type of parameter is the function parameter.

```
Example : 10  FUNCTION INTEG!(VAR X! FN Z!):LOCAL TOT!
          20  FOR X!=-1.0 TO 1.0 STEP 0.1
          30  TOT!=TOT!+0.1*Z!
          40  NEXT
          50  FN =TOT!
          60  END FN
          ...
          100 PRINT INTEG!(X!,SIN(X!))
          110 PRINT INTEG!(Y!,COS(Y!))
          ...
```

The function INTEG! calculates approximately the integral of a mathematical function in the interval [-1.0,1.0]. (The surface between the function and the X-axis.)
Note that the argument (X!,Y!) of the function (SIN(X!), COS(X!).) is transferred as a variable parameter.

4.6 LOCAL AND GLOBAL VARIABLES.

The variables in a procedure or function heading used to specifie value, variable, array or function parameters will be local variables, i.e. : after the procedure or function call has been executed these variables will keep the value they had before the procedure or function call.

```
Example : 10  PROCEDURE DUMMY I
          20  PRINT I:I=0
          30  END PROC
          ...
          100 I=10
          110 DUMMY 100
          120 PRINT I
```

In line 120 I will still have the value 10 after the procedure call DUMMY 100 in line 110.

However there is one exception on this rule.
When you call a procedure or function with variables, exactly the same as in the procedure or function heading to specifie variable and/or array parameters, these variables will be global.

```
Example : 10  PROCEDURE DUMMY VAR I
          20  PRINT I:I=0
          30  END PROC
          ...
          100 I=10
          110 DUMMY I
          120 PRINT I
```

In line 120 I will be 0, because it is considerd global to the procedure.

Also with the LOCAL statement you can define local variables.

```
Example : 10  PROCEDURE TEST:LOCAL J
          20  PRINT J:J=10
          30  END PROC
          ...
          100 J=1
          110 TEST
          120 PRINT J
```

In line 120 J will still have the value 1.

All other variables will be global to procedure or function.

```
Example : 10  PROCEDURE TEST
          20  J=10
          30  END PROC
          ...
          100 J=5
          110 TEST
          120 PRINT J
```

Due to the procedure call TEST, J will be changed to 10.

APPENDIX A

Summary of error-numbers and error-messages.

Error-number	Error-message.
0	NEXT WITHOUT FOR A variable in a NEXT stament does not corrospond to any previously executed, unmatched FOR statement.
1	RETURN WITHOUT GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
2	OUT OF DATA A READ statement is executed when there are no DATA statements with unread data remaining in the program.
3	OVERFLOW The result of a calculation is to large to be represented in DBASIC's number format. If underflow occurs, the result is zero and execution continues without an error.
4	UNDEFINED LINENUMBER The linenumber specified in this statement does not exist.
5	SUBSCRIPT ERROR An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts.
6	DIVISION BY ZERO A division by zero is encountered in an expression.
7	OUT OF STRING SPACE String variables have caused DBASIC to exceed the amount of free memory remaining in the heap.
8	STRING TOO LONG An attempt is made to create a string more than 255 characters long.
9	NUMBER OUT OF RANGE One of the arguments of the statement is not in the valid range.
10	INVALID NUMBER The parameter given to the VAL() function was not a valid floating point number.
11	LOADING ERROR 0 (cassette) Block length checksum error.
12	LOADING ERROR 1 (cassette) Insufficient memory.

13 LOADING ERROR 2 (cassette)
Block checksum error.

14 LOADING ERROR 3 (cassette)
Data drop-out error.

15 UNDEFINED ARRAY
An array has not been dimensioned before.

16 COLOUR NOT AVAILABLE
The colour is not available in a 4-color mode.

17 OFF SCREEN
The X- and Y- coordinates are not in the range [0,XMAX] and [0,YMAX].

18 ERROR LINE RUN
An attempt has been made to execute an erroneous line.

19 OUT OF MEMORY
No free memory left.

20 TYPE MISMATCH
A string variable name is assigned to a numeric value or vice versa.

21 LINE NUMBER OUT OF RANGE
The linenumber is out of the range [1-65535].

22 STACK OVERFLOW
No stack-memory left. Can occur by having to many FOR-NEXT loops nested or to many levels of GOSUB.

23 SYNTAX ERROR
A line contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).

24 COMMAND INVALID
A statement that is illegal in direct mode is entered as a direct mode command or a command that is illegal in a program is entered in a program line.

25 CANT' CONT
No continue of the program is possible.

26 LINE TO COMPLEX
The total length of the compiled line would exceed 128 bytes.

27 OUT OF MEMORY
See DBASIC error 19.

28 INCORRECT PARAMETER NUMBER
The number of parameters in the procedure- or function-caller does not correspond to the number of parameters in the procedure- or function defenition.

29 INVALID VARIABLE PARAMETER
The parameter in the procedure- or function-caller is not an unsubscripted variable reference.

30 INVALID ARRAY PARAMETER
The parameter in the procedure- or function-caller is not an array reference.

31 INVALID FUNCTION PARAMETER
The parameter in the procedure- or function-caller is a variable reference.

32 OFFSET OUT OF RANGE
Internal DBASIC error. This error cannot occur unless in the compiled code, procedures and functions are located behind 32k of the textbuffer.

33 CAN'T CLEAR
A CLEAR is not allowed because no relocation of the program is possible anymore.
This error will occur when trying to clear in a subroutine , a FOR-NEXT loop or a PROCEDURE.

34 INVALID EXTENSION
A DBASIC extension (see appendix D) was not found or in bad format.

35 UNDEFINED PROCEDURE
The called procedure has not been defined or, when code to define the procedure has been included in the program the program has not been compiled.

36 UNDEFINED LABEL
The label refered to in the statement has not been defined or the program has not been compiled.

37 WEND WITHOUT WHILE
A WEND statement is encountered for which there is no previous, unmatched WHILE statement.

38 UNTIL WITHOUT REPEAT
A UNTIL statement is encountered for which there is no previous, unmatched REPEAT statement.

39 MISSING ELSE
A IF statement is encountered without a matching ELSE statement.

40 MISSING END IF
A ELSE or IF statement is encountered without a matching END IF statement.

41 MISSING WEND
A WHILE statement is encountered without a matching WEND statement.

42 MISSING UNTIL
A REPEAT statement is encountered without a matching UNTIL statement.

43 MISSING FUNCTION TERMINATOR
A function defenition is encountered without at least one FN= statement.

44 MISSING END FN
A function defenition is encountered without a matching END FN statement.

45 MISSING END PROC
A procedure defenition is encountered without a matching END PROC

46 ELSE WITHOUT IF
A ELSE statement is encountered for which there is no previous, unmatched IF statement.

47 END IF WITHOUT IF
A END IF statement is encountered for which there is no previous, unmatched IF statement.

48 FUNCTION TERMINATOR WITHOUT FUNCTION
A FN = statement is encountered for which there is no previous, unmatched function defenition.

49 END FN WITHOUT FUNCTION
A END FN statement is encountered for which there is no previous, unmatched function defenition.

50 END PROC WITHOUT PROCEDURE
A END PROC statement is encountered for which there is no previous, unmatched procedure defenition.

51 FUNCTION TYPE MISMATCH
The expression type in the FN= statement is different from the type of the defined function or, the expression type in the function-caller does not correspond to the type of the function-parameter in the function defenition.

52 UNDEFINED COMMAND
The appropriate DBASIC extension to which the command belongs is not in memory.

53 SYMBOL IN USE.
The symbol used to define a label or procedure is already used for definition of a label or procedure. (duplicate definition)

54 INVALID USE OF FUNCTION
A function call is not allowed as argument of the specific function, usualy only a variable reference is allowed. Ex. : VARPTR(FAC%(N)) with FAC%(N) a 'user defined' function will generate the error.

55 RESUME WITHOUT ERROR
A RESUME statement is encountered without a previous occurance of an error.

APPENDIX B

Summary of compile-time errors

Note : For the exact meaning of the error-message refer to APPENDIX A.

1. Encoding errors : errors detected during encoding of the edit-buffer.

Error code	Error message
20	TYPE MISMATCH
21	LINE NUMBER OUT OF RANGE
23	SYNTAX ERROR
24	COMMAND INVALID
26	LINE TOO COMPLEX
27	OUT OF MEMORY

2. Compilation errors : errors detected during compilation. (See COMPILE)

37	WEND WITHOUT WHILE
38	UNTIL WITHOUT REPEAT
39	MISSING ELSE
40	MISSING END IF
41	MISSING WEND
42	MISSING UNTIL
43	MISSING FUNCTION TERMINATOR
44	MISSING END FN
45	MISSING END PROC
46	ELSE WITHOUT IF
47	END IF WITHOUT IF
49	END FUNCTION WITHOUT FUNCTION
50	END PROCEDURE WITHOUT PROCEDURE
52	UNDEFINED COMMAND
53	SYMBOL IN USE

APPENDIX C

Error reporting

- DBASIC has extended error reporting possibilities.
  Unless you did forsee an error trap routine, in runtime DBASIC will report known error-codes as shown in appendix A, eventually completed whith the message 'IN LINE nnnnn' during program execution.

- Compilation errors will be also reported with a 'IN LINE nnnnn' message.
  For error codes 37,38 and 46 to 53 nnnnn will give you the number of the line where the error has been detected.
  However for error codes 39 to 45 (MISSING ... errors) nnnnn will give you the number of the line in which a statement has been discovered, for which a neccessary matching statement could not be found.

  ex. : The error message 'MISSING WEND IN LINE 100' means that no matching WEND statement could be found for the WHILE statement in line 100.

- Untrapped user-defined error-codes will be reported as :

  'ERROR nnn [IN LINE nnnnn]'

  Ex. : *ERROR 100
  ERROR 100

- Besides DBASIC error-reporting, there is also an DBASIC-EXTENSION error-reporting.
  Thus extensions, like the Memocom MDCR driving package, can have their own error-codes and error-messages.

  Ex. : - The statement 'DCR 10' will generate the error-message 'DCR DRIVE NUMBER OUT OF RANGE'
  - The statement 'SKIP A%+1' with A%<-1 or A>255 will generate 'DCR FILE NUMBER OUT OF RANGE'.
  Etc ...

  See also appendix E : Memocom MDCR driving extension.

- When DBASIC extensions do not forsee special error-reporting routines, the errors will be reported as :

  '<extension name > ERROR nnn'

- For error-evaluation in error-trap routines three 'implicit-defined' functions are to the disposal of the user.

  1. The 'ERR' function gives the error-code of the last occured error.

     Ex. : If 'ERR=23' then the last occured error was a 'SYNTAX ERROR'.

  2. The 'ERL' function gives you the number of the line in which the error occured.

     Ex. : If 'ERL=1000' then the error occured in line 1000.
     If 'ERL=0' then the error occured in command mode.

  3. The 'ERR$' function gives you the name of the DBASIC-EXTENSION which generated the error.

     EX : If 'ERR$="DCR"' then the DCR driving extension generated the error.
     if 'ERR$=""' (null-string) then it is a DBASIC-error or a user defined error. ('ERR>55' for DBASIC V2.1)

APPENDIX D

$SYTEM EXTENSION

1. Description

$SYSTEM is a DBASIC extension, designed to extend DBASIC with other extensions and/or delete from DBASIC the extensions which are not needed anymore.

Extending DBASIC means : loading an extension-file ($-type), relocating the runtime-code, linking the extensions command-table to DBASIC and updating the BASIC-pointers. (Heap etc.)

Deleting from DBASIC means : masking out the extensions commnand-table and eventually releasing the memory occupied by the location if it is located just below the Heap.

2. COMMANDS

$SYSTEM contains two commands :

- $EXTEND

  Format : $EXTEND <string expression>

  Purpose : To extend DBASIC.

  Remarks : This command looks for the extension file with name equal to <string expression> and load it into memory starting at the Heap. The command table will be linked to DBASIC and the runtime-code will be relocated.

  Example : $EXTEND "DCR" will load and relocate the DCR driving extension.

- $DELETE

  Format : $DELETE <string expression>

  Purpose : To delete a DBASIC extension.

  Remarks : This command will search in memory for the extension with name equal to <string expression>. If the extension is in memory, the extension command-table will be masked out and if the extension is located just below the Heap, then the amount of memory occopied by the extension is released and BASIC-pointers are updated. $DELETE will let you know if everyting is done correctly by printing the message 'DONE'. If no extension with the name <string expression> could be found then $DELETE just prints a carriage return.

- Note : - The DBASIC program currently in memory will not be destroyed by $EXTEND or $DELETE, however if it was compiled, you will have to compile it again.

  - It is possible to delete the $SYSTEM-extension by giving the command : $DELETE "$SYSTEM"
    This gives you the smallest DBASIC-code and the largest amount of program-space. However you loose the possibility to use extra commands.

APPENDIX E

THE MEMOCOM-MDCR DRIVING PACKAGE

The Memocom-MDCR driving package is a DBASIC-extension. ($DCR file on DCR version of DBASIC V2.1)
The purpose of this extension is to integrate the DCR commands into DBASIC, and to forsee a correct error-reporting.

All the commands of the DCR-extension can be used both in program and direct-command mode.
Between the command and <integer expression> must be at least one space.

1. DCR

Format : DCR [<integer expression>]

Purpose : Selects DCR-drive <integer expression>.

Remarks : If <integer expression> is not in the range [0,3] a 'DCR DRIVE NUMBER OUT OF RANGE' error will be generated.
The default drive is 0.

Example : *FOR I=0 TO 3:DCR I:LOOK:NEXT

2. CAS

Format : CAS [<integer expression>]

Purpose : Selects audio-cassette <integer expression>.

Remarks : If <integer expression> is not in the range [0,3] a 'DCR DRIVE NUMBER OUT OF RANGE' error will be generated.
The default drive is 1.

Example : *CAS

3. REW

Format : REW [<integer expression>]

Purpose : To rewind the current DCR drive <integer expression> files.

Remarks : If <integer expression> is out of the range [0,255] a 'DCR FILE NUMBER OUT OF RANGE' error is generated.
The default is rewinding to the beginning of the tape.

Example : 10 DCR:REW 100

4. SKIP

Format : SKIP [<integer expression>]

Purpose : To skip <integer expression> files on the current DCR drive.
If <integer expression> is out of the range [0,255] a 'DCR FILE NUMBER OUT OF RANGE' error is generated.
The default is skipping to the end of the tape.

Example : SKIP 15+I

5. LOOK

Format : LOOK

Purpose : To display the type and name of the next file on the current DCR drive.

6. VER

Format : VER

Purpose : To verify the previous file on the current DCR drive.

7. DEL

Format : DEL

Purpose : To delete the next file s on the current DCR drive.

8. LAST

Format : LAST

Purpose : To specifie the previous file on the current DCR drive as the last file on the tape.

ERROR REPORTING

If the DCR-extension generates an error the ERR$ 'intrinsic' function is set to 'DCR'.

The DCR-extension knows eigth error-codes :

Error number	ERROR MESSAGE Meaning of the error.
1	DCR END OF TAPE ERROR During writing of a file the end of tape was reached.
2	DCR DOOR OPEN ERROR During writing of a file the cassette-door has been opened.
3	DCR BLOCK LENGTH CHECKSUM ERROR Cfr. 'LOADING ERROR 0' on audio-cassette.
4	DCR INSUFFICIENT MEMORY There is not enough free memory to load the file. Cfr. 'LOADING ERROR 1' on audio-cassette.
5	DCR BLOCK CHECKSUM ERROR Cfr. 'LOADING ERROR 2' on audio-cassette.
6	DCR DATA DROP OUT ERROR Cfr. 'LOADING ERROR 3' on audio-cassette.
7	DCR DRIVE NUMBER OUT OF RANGE The drive number specified in the DCR or CAS statement was not in the range [0,3].
8	DCR FILE NUMBER OUT OF RANGE The file number specified in the REW or SKIP statement was not in the range [0,255].

INDEX
