METANIC COMAL-80 and its documentation are copyrighted by METANIC ApS, DENMARK.

It is illegal to copy any of the software in this COMAL-80 software package onto cassette tape, disk or any other medium for any purpose other than personal convenience.

It is illegal to give away or to resell copies of any part of this METANIC COMAL-80 software package. Any unauthorized distribution of this product or any part thereof deprives the authors of their deserved royalties. METANIC ApS will take full legal recource against violators.

If you have any questions about these copyrights, please contact:

METANIC APS
KONGEVEJEN 177
DK-2830 VIRUM
DENMARK

(R) METANIC COMAL-80 is a registered trademark of METANIC ApS.

(R) CP/M is a registered trademark of Digital Research, Inc.

(R) Z-80 is a registered trademark of Zilog, Inc.

IT IS ONE THING TO COMMAND THE SHIP
QUITE ANOTHER TO WRITE THE CHARTS

An old proverb, written long before the world of the byte, the nanosecond and the interpreter; yet these words often came to mind as we worked on this manual.

Explaining something as complex as a high level language is as fraught with reefs as any sea-voyage, so it our hope that this book will allow you to chart your way through the intricacies of COMAL-80 with the minimum of effort.

We have had many suggestions and comments as a result of the first edition of this manual and, if the next edition is to be an even greater improvement, then we still need feedback from you, the user - the most important person of all.

There is an error report card at the back of this binder and you are invited to send any corrections, comments or suggestions that you think may be of use - we, in turn, will be happy to receive them. The format of the manual makes it very easy to update, so there is every chance that you will see your suggestions in print in a very short time.

An important part of the philosophy behind COMAL-80 is its ease of use, especially for those not necessarily familiar with high level languages. For this reason, and because this is a manual not a teaching book, all the keywords have been arranged in alphabetical order rather than in structural, but possibly unfamiliar, groups.

We hope you will come to find COMAL-80 an indispensible tool in your everyday computing and that this manual will help you to enjoy many pleasant and successful hours with your computer.

THE AUTHORS

METANIC ApS hereby wishes to thank the following members of the staff and friends of COMAL-80 for their dedicated assistance in the preparation of this manual:

MOGENS PELLE
ARNE CHRISTENSEN
MOGENS CHRISTENSEN
SUSANNE SONDERSTRUP

A special acknowledgement is extended to all the pioneers who helped with field testing the COMAL-80 interpreter, and whose criticism and suggestions have had so much impact on the final specifications.

FOURTH EDITION, JUNE 1983.
GENERAL CP/M VERSION
PRINTED IN DENMARK

METANIC COMAL-80, written for the Z-80 microprocessor, is the most extensive interpreter available for microcomputers today and contains, as well as a full extended BASIC, a great number of structures found in Pascal.

COMAL-80 was originally specified as a result of the specific wishes of Danish educationalists who wanted an easy to learn language with built-in programming support which would facilitate transition to other structured languages.

This manual is divided into two parts with a number of appendices.

Part 1 contains instructions for initialization of the different versions of COMAL-80 and a general description of features which affect some or all the COMAL-80 instructions.

Part 2 contains the syntax and semantics of all commands, statements and functions in alphabetical order.

The appendices contain the source code for the screen driver, guidelines for changing this to suit different systems, a list of error messages, demonstration programs and a list of ASCII codes.

This manual is not intended as a tutorial for COMAL-80, but as a reference manual to the specific features of METANIC COMAL-80.

Each of the two COMAL-80 software packages contains two versions of the COMAL-80 interpreter. The two versions have identical features, except that the overlayed version leaves more storage for the user while requiring a few seconds at the start and end of each program execution to read the overlay file.

The different files are named:

7-digits precision:

Non-overlayed version:	COMAL-80.COM
Overlayed version:	COMAL80S.COM
Overlay file:	COMAL-80.2

13-digits precision:

Non-overlayed version:	COMAL80D.COM
Overlayed version:	CMAL80DS.COM
Overlay file:	COMAL80D.2

Note that each package contains the files for only one of the two possible precisions and that the CP/M operating system is not included on the distribution disks.

It is suggested that the COMAL-80 files be copied to a new disk together with the CP/M operating system. Then remove the original disk from the computer and keep it in a safe place as this disk alone carries a warranty.

Now type the name of the version without the extension '.COM' and COMAL-80 will sign on. Note that the overlay versions will work only if the disk is placed in the CP/M default drive.

Once initialized, COMAL-80 checks whether an initialization file exists on the disk. If so, it is read and executed. This file is described in detail in a later chapter. If no such file exist, COMAL-80 simply asks whether error descriptions are required. Answer with 'Y' for yes or 'N' for no.

COMAL-80 is then ready for use, as shown by the prompt character '*'. Commands and program statements may then be keyed in.

Commands are recognized by the fact that they do not start with a line number. A command will be executed immediately following a 'RETURN'.

The special system commands (such as 'RUN', 'LIST', etc.) as well as many of the COMAL-80 statements may be used as commands allowing instant results of arithmetic and logical operations to be displayed without any need to write a program.

Program statements are recognized by the fact that they start with a line number. This indicates to COMAL-80 that the line should be stored for later execution.

On pressing 'RETURN', a line is syntax-checked and if no errors are found it is converted to an internal format and stored in the working memory of the computer. If an error is found, the line is displayed on the terminal with the cursor indicating the error point. An error code and, if the error descriptions are not deleted, a description of the error are also displayed.

Using the editing facilities of COMAL-80, the error may then be corrected and followed with 'RETURN'. The above sequence is then repeated until the line is correct.

When the user types 'RUN' a prepass is executed first to complete the translation into internal format. Among other things it translates all references to absolute memory addresses.

Finally the run module goes into action to execute the program.

The statement lines in COMAL-80 have the following format:

```
nnnn COMAL-80 statement [//(comment)]
```

nnnn is a line number between 1 and 9999. Only one statement is allowed on each line unless separated by semicolons. For further details see the 'LET' and 'MAT' statements.

All statements may be followed by a comment (see also 'REM' in chapter 2).

A COMAL-80 statement always starts with a line number, ends with 'RETURN', and may contain up to a maximum of 159 characters. On terminals with a physical line length less than this, a line, once filled will be continued on the next screen line.

INPUT EDITING

If an error is made while a line is being typed in, move the cursor back to point at the error and type the correct character(s). The new character(s) will replace the old one(s). The character pointed at by the cursor can be deleted by pressing the 'DEL' key (user defineable) whereupon all characters to the right of the cursor will move one position left.

New characters may be inserted between existing characters by moving the cursor to the position where the insert is to start and pressing the 'INS' key (user defineable). The rest of the line (including the character pointed at by the cursor) will move one position to the right leaving an empty space. This can be repeated as often as necessary to create space for any number of characters up to the maximum line length of 159 characters.

When the input is terminated by pressing the 'RETURN' key, the whole line shown on the screen is stored regardless of the cursor position.

A line which is in the process of being typed may be deleted by pressing the 'ESC' key (user defineable). This will also terminate the automatic generation of line numbers.

To correct program lines of a program which is currently in memory, re-type the line using the same line number or use the 'EDIT' command.

To delete an entire program currently residing in memory use the 'NEW' command.

The COMAL-80 character set comprises the alphabetic characters, numeric characters and special characters.

The alphabetic characters are the upper and lower case letters of the alphabet, including { | } [\ ] which may be replaced by national letters in some countries.

The numeric characters are the digits 0 through 9.

The following special characters are recognized by COMAL-80:

CHARACTER	NAME
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Multiplication symbol
/	Slash or division symbol
^	Exponentiation symbol
(	Left parenthesis
)	Right parenthesis
#	Number or hash sign
$	Dollar sign
!	Exclamation point
,	Comma
.	Period or decimal point
"	Double quotation marks
;	Semicolon
:	Colon
&	Ampersand
<	Less than
>	Greater than
_	Underscore
'ESC'	* Stop and wait for input
'RETURN'	Terminate input
Control-A	* Insert
Control-H and <=	* Cursor left
Control-L and =>	* Cursor right
Control-S	* Delete
Control-K	* Cursor to start of line
Control-J	* Cursor to end of line
Control-I	* Cursor 8 steps forward
Control-B	* Cursor 8 steps backwards
Control-E	* Delete to end of line

* user definable.

Constants are the actual values which COMAL-80 uses during execution. There are two types of constants: string and arithmetic.

A string constant is a sequence of alphanumeric characters enclosed in double quotation marks. The length of the string is limited only by the space available in the computer.

A double quotation mark may be included in a string constant by entering 2 double quotation marks ("") immediately following each other.

Characters which cannot be typed on the keyboard, can be included in a string constant by typing the characters' decimal ASCII code enclosed in double quotation marks.

EXAMPLES OF STRING CONSTANTS:

```
"COMAL-80"
"$10.000"
"OPEN THAT DOOR"
"KEY ""S"" TO STOP"
"END"13""
```

Arithmetic constants are positive and negative numbers. Arithmetic constants in COMAL-80 cannot contain commas. There are two types of arithmetic constants:

1. Integer constants — Whole numbers in the range -32767 to 32767. Integer constants do not contain a decimal point.

2. Real constants — Positive or negative real numbers, i.e. numbers that contain a decimal point and positive or negative numbers represented in exponential form (scientific notation). A real constant in exponential form consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter 'E' and an optionally signed integer (the exponent). In addition, whole numbers outside the range for integer constants are considered to be real constants.

Variables are names used to represent values used in a COMAL-80 program. The value of a variable may be assigned explicitly by the programmer or it may be assigned as the result of calculations in the program. Until a variable has been assigned a value, it is undefined.

VARIABLE NAMES AND DECLARATION CHARACTERS

COMAL-80 variable names may be of any length up to 80 characters. The characters allowed in a variable name include all letters, digits and the underscore. The first character must be a letter. Special type declaration characters are also allowed. - See below.

A variable name may not be a reserved word unless the reserved word is embedded. Reserved words include all COMAL-80 commands, statements, function names, operator names and identifiers defined in an "EXTENSION".

Variables may represent either an arithmetic value or a string. String variable names are written with a '$' (dollar sign) as the last character. Integer variable names are written with a '#' (number or hash sign) as the last character. The '$' and the '#' signs are variable type declaration characters, i.e. they 'declare' that the variable will represent a string or an integer.

Examples of variable names:

```
A
A8
DISKNAME$
COUNTER#
VALUE_OF_CURRENT
```

An array is a group or table of values referenced by a single variable name. Each element in an array is referenced by a variable name subscripted with one arithmetic expression for each dimension. An array variable name has as many subscripts as there are dimensions in the array. When used as a parameter the array can be referenced as a whole or as an 'array of arrays' by omitting some or all the subscripts. This is described in detail in the chapter: PARAMETER SUBSTITUTION.

All arrays must be declared using a 'DIM' statement or a 'RECEIVE' statement.

When an arithmetic array is declared, but before it has been assigned any values, all its elements have the value 0 (zero).

When a string array is declared, but before it is assigned strings, all its elements contain the string "" (string of zero length).

SUBSTRINGS

As well as from referencing a string variable as a whole, or (for arrays) element by element, or as an array of arrays, a part of a string variable element may also be referenced.

This is done in one of the following formats:

```
⟨name⟩(I1,I2,...In,⟨start⟩[,⟨end⟩])
⟨name⟩(I1,I2,...In)(⟨start⟩[:⟨end⟩])
```

In the first case, the number of dimensions in the variable ⟨name⟩ is retrieved from the corresponding 'DIM' statement. If it has, say 'n' dimensons, then the first 'n' indices in the parenthesis are used to specify the actual element. The parenthesis may contain one or two further indices, i.e. ⟨start⟩ and ⟨end⟩. ⟨start⟩ specifies at which character position the substring starts, and ⟨end⟩ specifies where it ends. Whithout ⟨end⟩, the substring consists of the character at the ⟨start⟩ position only.

In the second case, the first parenthesis contains the necessary number of indices, whereas the second parenthesis contains ⟨start⟩ and ⟨end⟩ information as described before.

If ⟨name⟩ states a simple string variable then the number of dimensions is considered to be zero and the parenthesis contains ⟨start⟩ and ⟨end⟩ only. In the latter format, the first parenthesis is omitted.

The arithmetic operators are:

Precedence	Operator	Operation	Example
1	^	Exponentiation	X^Y
2	/	Division	X/Y
2	*	Multiplication	X*Y
2	DIV	Integer division	X DIV Y
2	MOD	Modulus	X MOD Y
3	-	Negation	-X
3	+	Addition	X+Y
3	-	Subtraction	X-Y

Precedence controls the order in which operations are handled within an expression. The operator with the highest precedence is evaluated first, lowest last. Where several operators have the same precedence they will be evaluated from left to right.

Precedence may be overruled by parentheses: expressions enclosed in parentheses are resolved first. When multiple operators occur in the same set of parentheses the above table applies.

Apart from negation, the arithmetic operators may be used only between expressions giving arithmetic values. Negation may be used only for expressions giving arithmetic values.

The arithmetic value of a logical true expression is 1. The arithmetic value for a logical false expression is 0.

Relational operators are used to compare two values. The result of a such comparison may be either true (= 1) or false (= 0). This result may then be used to influence the program run.

Whenever an arithmetic value is used as a logical value, the number 0 is interpreted as false, and numbers other than 0 are interpreted as true.

Operator	Relation	Example
=	Equality	X=Y
<>	Inequality	X<>Y
>	Greater than	X>Y
<	Less than	X<Y
>=	Greater than or equal to	X>=Y
<=	Less than or equal to	X<=Y

(= is also used to assign a value to a variable.)

Relational operators are used between two expressions both giving an arithmetic value or between two expressions both giving a string value.

Relational operators have a lower precedence than arithmetic operators. Within an expression containing both types all arithmetic operators are resolved before the relational operators.

In the following example:

```
X-2>T+3
```

the values of 'X-2' and 'T+3' are calculated before the comparison of the two values.

Comparison between two string expressions is performed character by character using the ASCII codes for each character. 'A' is less than 'E' (the ASCII code for 'A' is 65 and for 'E' is 69).

With two strings of different lengths where the short one is equal to the beginning of the long one, the short one is considered the smallest. Consequently, "BLACK" is smaller than "BLACKBIRD".

When comparing two strings, all characters between the double quotation marks are compared including spaces. In this respect the aggregates "" and "number", each representing only one character when found within a string value, count as one character only, namely the character represented by the aggregate.

File names basically follow the CP/M naming conventions. Only the first eight characters are significant and lower case letters are converted to upper case.

Following a period an extension of three characters may be specified. The extension can be chosen freely except in connection with 'SAVE' and 'LOAD' commands where the COMAL-80 system automatically provides the extension '.CSB'. No extension may be specified in construction with these commands.

If no extension is specified, the default '.CML' is used whenever the file name is used in connection with the 'ENTER' and 'LIST' commands, '.DAT' is used in connection with the 'OPEN' command/statement, except for random files where 'RAN' is used. '.CAT' with the 'CAT' command/statement and '.LOG' is used for log files.

The whole name, including the extension, is used to specify a file. This means that the two commands:

```
ENTER PROGRAM
ENTER PROGRAM.CML
```

read the same file into memory, whereas

```
ENTER PROGRAM.LST
```

reads another.

The disk drive name is optional but is treated as an integral part of the file name. If it is omitted, the current default disk drive is used. If it is specified then it is written in front of the file name. The disk drive name is the device name of the disk to be used (see below).

Example:

```
ENTER DK1:PROGRAM.CML
```

Note that the disk drive names do not follow the CP/M naming convention.

The disk drive name consists of the two letters 'DK' (meaning disk) and a unit number followed by a colon. Thus 'DK0:' corresponds to CP/M's 'A:', 'DK1:' corresponds to CP/M's 'B:', etc.

A similar system is used with the other peripheral devices, so that these can be used as files and may be the source of or destination for, data according to the nature of the specific device.

The names used for the different devices are:

```
'LP:' or 'LP0:' for the line printer
'LP1:'          for the punch device
'DS:' or 'DS0:' for the data screen
'KB:' or 'KB0:' for the keyboard
```

Example:

```
10 OPEN FILE 0, "KB:", READ
20 OPEN FILE 1, "LP:", WRITE
30 DIM A$ OF 100
40 LOOP
50  INPUT FILE 0:A$
60  PRINT FILE 1:A$
70 ENDLOOP
```

When 'INIT', 'RELEASE', 'LOG', 'DELETE', 'GETUNIT', 'RENAME', 'UNIT', and 'CAT' are used as statements, filenames are considered to be string expressions and must be enclosed in double quotation marks. This is optional in command mode. This allows a file name to be specified by any string expression which evaluates to a legal file name.

Examples:

```
100 DELETE "DK0:PROGRAM.CML"
100 INIT "DK0:",A$
100 DELETE "DK0:"+A$+".CML"
```

COMAL-80 uses its own format in disk files. The normal CP/M format can be specified by extending the filename with a '/C'. Further extending the filename with a '/B' specifies the CP/M binary format.

Examples:

```
    ENTER TEST.BAK/C                    // READ CP/M ASCII FILE
100 OPEN FILE 3,"TEST.XYZ/C/B",READ //OPEN CP/M BINARY FILE
100 OPEN FILE 2,"DATA/C",WRITE      //OPEN CP/M ASCII FILE
```

One of the distinct features of COMAL-80 is the inclusion of genuine procedures with parameters.

A procedure is a named program area placed between the keywords 'PROC (name)' and 'ENDPROC (name)' and which is called by the use of the keyword 'EXEC (name)'.

They act basically as subroutines and can be called from one or several places within a program. When the procedure has been completed the program execution continues on the line following the calling line. Apart from this they have other features which make them a very efficient programming tool.

Firstly, they are called by name so that the programmer does not have to worry about the line numbers at which the procedure is located.

Secondly, the procedure is non-executable until it is called, meaning that regardless of where the procedure is placed in the program, the lines inside it will be bypassed unless the procedure is actually called by an 'EXEC' statement. This call can go forwards or backwards in a program.

Thirdly, and very important, parameters can be passed to the procedure when it is called. This means that a procedure can react differently and operate on different data each time it is called.

There are two types of procedures, called open and closed procedures. The difference between the two is a question of how the procedure sees the variables used in the rest of the program.

A variable used in an open procedure has the same status as a variable used in the main program. This means that if it is assigned a new value within a procedure, it keeps this value when the procedure is terminated and program execution resumes from the line following the calling line.

The closed procedure, however, acts in many ways like a separate program. The closed procedure has its own set of variables, which can be dimensioned and assigned values within the procedure, but they are never able to influence the variables used outside the procedure unless some special action is taken (refer to parameters and the import statement). This makes it possible to write library routines which can be used in any program without risking problems with the same variable name being used both in the procedure and in the rest of the program.

The difference between the two types of procedures can be illustrated within the following two programs:

```
1                          2

10 A:=5                    10 A:=5
20 EXEC TEST               20 EXEC TEST
30 PRINT A                 30 PRINT A
40 PROC TEST               40 PROC TEST CLOSED
50  A:=3                   50  A:=3
60  PRINT A                60  PRINT A
70 ENDPROC TEST            70 ENDPROC TEST
```

Running these 2 programs the first one will print the digit '3' twice because the assignment in line 50 will overrule the assignment in line 10. The second example will print the digits '3' and '5' because the procedure is closed and therefore the variable in line 50 is not the same as the one in line 10 - even though they have the same name. Technically speaking, the variable 'A' in example 1 is global because the whole program can see and use it, but a variable inside a closed procedure is local and can only be used inside the procedure.

A local variable must also be assigned (line 50) or dimensioned inside the closed procedure before it is used for the first time. This means that if line 50 is deleted in the second example, the program execution will stop in line 60 with an error message saying that the variable is unknown.

Even though the separation of variable names is the basic idea behind the closed procedures, it is often convenient to make a variable name known to the main program as well as to the procedure

This can be done through the 'IMPORT' statement as shown in the following example:

```
10 A:=3
20 EXEC TEST
30 PRINT A
40 PROC TEST CLOSED
50  IMPORT A
60  A:=3*A
70  PRINT A
80 ENDPROC TEST
```

This program will print the digit '9' twice. Note that the 'IMPORT' statement must be placed in the closed procedure and before the part of the procedure actually using the variable for the first time.

Closed procedures can be nested to any level that the memory allows but the 'IMPORT' statement only works at the level where it is actually placed. The following program will print the digit '3' (in line 100) and then stop in line 60 with an error message that the variable is unknown:

```
 10 A:=3
 20 EXEC TEST1
 30 PRINT A
 40 PROC TEST1 CLOSED
 50  EXEC TEST2
 60  PRINT A
 70 ENDPROC TEST1
 80 PROC TEST2 CLOSED
 90  IMPORT A
100  PRINT A
110 ENDPROC TEST2
```

Another way of moving a variable into and out of a closed procedure is by means of a reference parameter. this is described in details in the chapter 'PARAMETER SUBSTITUTION'.

When a variable is dimensioned or assigned a value in a closed procedure the necessary memory is not allocated until the procedure is actually called and this memory is again de-allocated when the procedure is terminated.

Thus, no matter how many of times a procedure is called, there will be no error message 'out of storage' as long as it does not happen on the first call.

This also makes it possible to dimension a variable in a procedure which is called several times without conflicting with the rule that a variable cannot be re-dimensioned, and it is possible to overlay arrays and string variables used for intermediate results and thereby economize on storage by dimensioning and using these in different closed procedures.

Any procedure may call any procedure defined anywhere in the main program and it may even call itself (recursion). However, a closed procedure can only call a closed procedure. Note also, that recursion implies nesting to a new level which uses memory and must be carefully controlled.

The rules for variables in closed procedures also apply to the other closed structure: The user-defined function.

An important part of the COMAL-80 definition is the inclusion of procedures (and user-defined functions) with parameters, which allow a program to be broken down into smaller, named routines. These can be open or closed.

To move data into and out of a such routine parameters are used, i.e. list of variable names specified in the first line of the routine (the formal parameters) and a list of variables or expressions in the calling line (the actual parameters). The actual parameters are then inserted into the formal parameters when the routine is called.

There are two types of parameters, namely 'call by value' and 'call by reference'.

'call by value' means that the actual value of the actual parameter is assigned to the formal parameter. This type can only move data into the routine as changes to the formal parameter do not affect the actual parameter.

'call by reference' means that the formal parameter is replaced by the actual parameter. This type can move data both into and out of a routine, and is specified by the keyword 'REF' in the formal parameter list. The above mentioned replacement happens dynamically, i.e. when the routine is called, and it cannot be seen in program listings which always show the formal parameters.

The following examples show the difference:

```
1                          2
10 A:=3                    10 A:=3
20 EXEC TEST(A)            20 EXEC TEST(A)
30 PRINT A                 30 PRINT A
40 PROC TEST(X)            40 PROC TEST(REF X)
50  X:=3*X                 50  X:=3*X
60  PRINT X                60  PRINT X
70 ENDPROC TEST            70 ENDPROC TEST
```

Here, in line 20 'A' is the actual parameter and 'X' in line 40 is the formal parameter.

In the first example the value '3' is assigned to 'X' when the procedure 'TEST' is called in line 20 and prints the digit '9' in line 60. After the procedure is terminated the digit '3' is printed in line 30 because the variable 'A' is in no way affected.

The other example will print the digit '9' twice because the formal parameter is replaced by the actual one and the change is thereby reflected back.

Parameters are always local, meaning that changes which happen to 'call by value' parameters in a routine cannot affect a variable of the same name in the rest of the program. This is shown by the following example:

```
10 A:=3
20 B:=2
30 EXEC TEST(A)
40 PRINT A,B
50 PROC TEST(A)
60  A:=3*A
70  B:=3*B
80  PRINT A,B
90 ENDPROC TEST
```

For 'A' this program will print the digit '9' in line 80 and then the digit '3' in line 40. Both lines print the digit '6' as the value for 'B'. In other words, the formal parameter 'A' is local to the procedure and another variable than the variable used in lines 10 and 40, whereas 'B' is not a parameter (and the procedure is not closed) so it is global to the procedure, and the same variable in the whole program.

The parameter lists may contain as many parameters as the maximum line length allows (159 characters), separated by commas, but there must be the same number of parameters in both lists, and corresponding parameters must conform to type and dimension. The only exception is that an integer actual parameter can be assigned to a real formal parameter when 'call by value' is used.

Constants and expressions can be used as actual parameters when 'call by value' is used.

Example:

```
10 EXEC TEST(3*5,"ERROR")
20 PROC TEST(A,B$)
30  PRINT A
40  PRINT B$
50 ENDPROC TEST
```

Note, that a formal parameter cannot be dimensioned, since the call itself carries the necessary information.

Arrays can be used as parameters either as a whole, as an array of arrays or as a single element, but they can only be used as reference parameters in the former two cases.

When a single element is used, the element is specified in the actual parameter list with the necessary number of indices and a variable of the same type specified in the formal parameter list.

Example:

```
 10 DIM A(3,5,2)
  .
  .
100 EXEC TEST(A(1,1,1))
  .
  .
200 PROC TEST(B)
  .
  .
300 ENDPROC TEST
```

Note, that 'B' does not need to be a referenced parameter since only a single element is used.

An array of arrays is used by omitting one or several of the indices from the right hand side in the actual parameter list and following the formal parameter name with a parenthesis containing the same number of commas as the number of omitted indices minus 1.

Example:

```
 10 DIM A(3,5,2)
  .
  .
100 EXEC TEST(A(1,1))
  .
  .
200 PROC TEST(REF B())
  .
  .
300 ENDPROC TEST
```

In this example one should note that the parenthesis following the formal parameter 'B' is empty because the number of omitted indices is 1.

The omitted indices are then specified when the formal parameter is used in the routine.

The following example shows this:

```
 10 DIM ARRAY_OF_VECTORS(5,3)
 20 FOR I:=1 TO 5
 30  FOR J:=1 TO 3
 40   ARRAY_OF_VECTORS(I,J):=RND(1,5)
 50  NEXT J
 60 NEXT I
 70 EXEC CHANGE_SIGN(ARRAY_OF_VECTORS(4))
 80 PROC CHANGE_SIGN(REF VECTOR()) CLOSED
 90  FOR I:=1 TO 3
100   VECTOR(I):=-VECTOR(I)
110  NEXT I
120 ENDPROC CHANGE_SIGN
130 FOR I:=1 TO 5
140  FOR J:=1 TO 3
150   PRINT ARRAY_OF_VECTORS(I,J);
160  NEXT J
170  PRINT
180 NEXT I
```

It is also possible to use a whole array as a parameter. This is done by removing all the indices in the actual parameter list and following the formal parameter with a parenthesis containing the same number of commas as the dimension of the array minus 1.

Example:

```
 10 DIM A$(5,3,2) OF 25
  .
  .
100 EXEC TEST(A$)
  .
  .
200 PROC TEST(REF B$(,,))
  .
  .
300 ENDPROC TEST
```

COMAL-80 actually consists of 3 main modules called:

Input module
Prepass module
Run module

Each module has its own error routines handling different error types as efficiently as possible.

These routines have at their disposal a library of error messages each giving a short description of about 200 different types of errors.

An error number is always given with the error message and in most cases the actual line causing the error is displayed with the cursor indicating the point of error.

To give instant error messages the library is an integrated part of COMAL-80. The library uses about 3K and it is possible to delete most of it when signing on COMAL-80, giving the user about 2.5K extra storage.

Except for the missing messages, the rest of the error reporting system works in the usual way and the error number makes it possible to find the text by referring to appendix C of this manual.

SYNTAX ERRORS.

The input module consists of two submodules: the editor and the syntax controller.

The editor is a line-oriented editor, which allows the user to key-in a line and change it as appropriate. When the line is terminated by pressing (return) it is transferred to the syntax controller and checked against COMAL-80 specifications.

If no syntax errors are found the line is executed (if it is a command) and translated and stored in memory (if it is a statement).

If the line contains a syntax error, an error number and (if available) an error message is displayed followed by the actual line with the cursor indicating the error location, control is then returned to the editor. The user can then correct the line and repeat the sequence until the line is accepted.

When reading an ASCII file via the 'ENTER' command, each line is syntax checked in the same way. If an error occurs the line is displayed on the console together with an error number (and possibly a message) and an arrow pointing to the error, the line is not stored in memory. Loading is then resumed.

It is not possible to store a line containing a syntax error.

PREPASS ERRORS

When the user wants to execute a program and types 'RUN' the prepass, which is invisible to the user, goes into action. This module extends the internal representation of the program using absolute memory addresses and checks that all structures are properly terminated and that all reference points exist.

If no error is found, control is passed on to the run module.

If one of the statements of a structure is missing (FOR...NEXT, REPEAT...UNTIL, WHILE...ENDWHILE, a.s.o.), the line number of the corresponding statement is displayed on the screen with an error number and possibly an error message. Line numbers with calls to non-existing 'LABEL' statements are shown in the same way.

If a statement contains the 'EXIT' statement without the surrounding 'LOOP' and 'ENDLOOP' statements, the line number of the 'EXIT' statement is returned.

All errors in a program are reported at the same time, and control is then returned to the input module. It is not possible to execute any part of a program if it contains a prepass error.

RUN ERRORS

When the run module is called only errors of dynamic nature (i.e. occurring when a line is actually executed) can exist. An error of this type will normally stop COMAL-80. The line containing the error will be shown on the screen with the error number and, possibly, an error message. Control is then returned to the editor in the input module for easy correction of the error. However, a number of errors are non-fatal because they can be bypassed in a well-defined manner. An example of this is division by 0 where it is often convenient to assign as the result the maximum value that COMAL-80 can handle.

To prevent a program stopping for non-fatal errors, two special statements are implemented: 'TRAP ERR-' and 'TRAP ERR+'.

If a 'TRAP ERR-' statement has been executed a non-fatal error will not stop the program execution, but a subsequent call of the system function 'ERR()' will return the error number. By testing this function it is then possible to influence program flow. This mode of operation continues until a 'TRAP ERR+' statement is executed after which the system returns to normal error handling.

The fatal errors always terminate program execution.

Note that the 'TRAP ERR-' mode is a question of having executed a such statement. Its actual line number is of no importance.

The 'RUN' command always resets to normal error handling.

COMAL-80 allows the same string of characters to be used for different variable types and for procedure and function names. To avoid conflicts, the following rules apply:

Static binding is used everywhere. It must be clear from the program text what each name stands for. This is reflected by several of the rules below.

All variables introduced within a closed procedure or function are local to the procedure or function and cannot be referenced from within any other procedure or function, or from the main program. This also applies to the parameters of a closed procedure/function since these are cases of local variables.

From within a closed procedure/function the following variables can be referenced: The local variables of the procedure/function and any variables explicitly imported from the main program by means of the 'IMPORT' statement. No other variables can be referenced from within the closed procedure/function.

Parameters to an open procedure/function are local to the procedure /function and cannot be referenced from within any other procedure/ function (not even from within another open procedure/function) or by the main program. An open procedure/function has no local variables apart from the parameters.

From within an open procedure/function the following variables can be referenced: The parameters of the procedure/function and the variables of the main program. No other variables can be referenced from within the open procedure/function. If any variables are 'DIM'ensioned within the open procedure they are, in all respects, treated as if they had been 'DIM'ensioned within the main program. Variables that are used within the open procedure/function but not in the main program, are still considered belonging to the main program.

The 'IMPORT' statement can only be used in closed procedures/functions. It cannot be used in an open procedure/function or in the main program.

System variables and functions can be referenced everywhere.

Procedures and functions cannot be nested, whether open or closed.

Procedures and functions can call each other and can call themselves recursively, whether open or closed.

Procedures can be called throughout the program. Procedure names cannot be 'IMPORT'ed.

Functions can be called throughout the program except at points where a variable with the same name masks the function. This is only possible if the function has parameters. The rule is: If a 'DIM'ensioned variable with the same name as the function can be referenced at a certain point then the function cannot be called at this point. Any attempt to call the function at this point will be interpreted as a reference to the variable with the parameters interpreted as indices.

A call to a function without parameters is programmed by writing the name of the function followed by an empty parenthesis, e.g. 'FUNCTION()'. Thus there is no possibility of confusion even if a variable with the same name can be referenced at the point of the call.

The above rules are introduced in order to allow for arbitrary naming of local variables in library procedures (which will always be closed).

Function names cannot be 'IMPORT'ed.

Labels defined in the main program can only be referenced in the main program. Labels defined in a procedure or function, whether it be open or closed, can only be referenced in the procedure or function where they have been defined. Labels cannot be 'IMPORT'ed.

After COMAL-80 has been read into memory and has started to execute (but before the prompt sign is displayed), it is often convenient to be able to change default values automatically, or to load and execute a program.

The initialization file offers this opportunity. Before the prompt sign is displayed, the CP/M default disk drive is searched for an appropriate file which is read in an executed (if it exists).

The file itself is a normal COMAL-80 text file stored on the disk with the 'LIST' command under the name 'COMAL80I.NIT' (for the 7-digit version) and 'CMAL80DI.NIT' (for the 13-digit version). As this is a text file, each line (except the very first one) follows the syntax:

```
(line No.) // (COMAL-80 statement or command)
```

The first line follows the syntax:

```
(line No.) // (error text mode) [(highest memory address)]
```

(line No.) is a normal line number used only when editing the initialization file.

The remark sign ('//') allows anything to be written on the rest of the line.

(COMAL-80 statement or command) is a normal COMAL-80 statement or command as described in chapter 2 of this manual; the only exceptions being that the 'AUTO' and 'EDIT' commands are not allowed.

(error text mode) specifies whether error texts are wanted or not. Possible answers are 'Y' for yes, 'N' for no, and 'A' for ask. When using 'A' the question whether or not error messages are wanted will be displayed on the screen.

(highest memory address) specifies in decimal the highest memory address for use by the COMAL-80 system. This specificaton is optional and defaults to the first memory position below the CP/M operating system.

When the initialization file is executed it is read line by line from the disk. The (line No.) and the remark sign are skipped and the rest of the line is executed exactly as a normal COMAL-80 line.

Example:

```
0010 // Y 50000
0020 // ZONE:=8; PAGEWIDTH:=132; PAGELENGTH:=0
0030 // CLEAR
0040 // PRINT "WELCOME TO COMAL-80"
```

This file contains commands only and in line 0010 the COMAL-80 system is instructed that error messages are wanted and that no memory location above address 50000 may be used.

The rest of the lines are then read and executed one by one.

The initialization file may also include statements as shown in the following example:

```
0010 // N
0020 // 10 // PRINT 80 STARS
0030 // 20 FOR I=1 TO 80
0040 // 30 PRINT "*",
0050 // 40 NEXT I
0060 // RUN
```

In this example the 'FOR...NEXT' loop is first stored in memory, controlled by the line numbers 20, 30, and 40 and then executed when the 'RUN' command is met in line 0060. Note, that the statements are actually stored in memory and they will still be there when the initialization file is terminated and the prompt sign is shown, unless a 'NEW' command is included.

It is also possible to load and execute one or more files:

```
0010 // N
0020 // LOAD DK1:PROG1
0030 // RUN
0040 // NEW
0050 // ENTER PROG2
0060 // RUN
```

This example will first load 'PROG1' from disk drive 1, execute it, clear the memory, and then it will enter 'PROG2' and execute that one.

To guarantee that the whole initialization file is executed the 'ESC' key is disabled until the last line is reached. This means when 'PROG1' is executing in the former example, it cannot be stopped by pressing the 'ESC' key, whereas the system reacts normally when 'PROG2' is running because the 'RUN' command is the last line in the initialization file.

The error checking system works in the normal way but due to the disabling of the 'ESC' key it is only possible to correct the line shown on the screen. This means that there are situations where errors can only be bypassed by deleting the whole line.

COMAL-80 offers extensions as a unique feature which allows the user to customize the language and extend to it by adding new statement types, standard functions and operators.

New keywords and the necessary Z80 machine code is stored in a disk file produced using a re-locatable assembler. One or more of these files are then activated using the 'EXTENSION' command, and the new keywords become reserved words which, in all aspects, act like the original keywords.

Appendix D shows a full working example of this feature which should be studied in conjunction with this description.

An assembler program defining extensions must start with

```
NAME('<name>')
```

where <name> is the name of the program.

After this, the contents of the file 'EXTDEFS.MAC' are followed with the specifications for the extensions defined in the file which must follow immediately after each other. Each extension will be formatted as follows:

```
EXTENSION <name>[,<local name>]
<interface>
ENDEXT    <name>[,<local name>]
```

The same <name> and, if used, <local name> must be used in both places. For the very last specification, the word 'ENDEXT' is replaced with 'ENDALLEXT'.

<name> is the name with which the extension will be called in a COMAL-80 program, excluding any possible dollar-sign.

<local name> is the name of the extension within the assembler file. If <local name> is not given, <name> will be used in the assembler file as well. It may be useful to give a <local name> if several extensions have names in which the first five characters are the same, or if <name> contains an underscore, or is the same as a Z80 op-code.

<interface> is different for functions, statements, and operators.

For functions:

```
FUNCTION  <return type>
<list of parameters>
```

where the <return type> is of either type 'INT', 'REAL' or 'STR'.

For statements:

```
STATEMENT
(list of parameters)
```

(list of parameters) must consist of zero or more lines with each line specifying one parameter in the following way:

```
PARAMETER [(dimension),](type)
```

If (dimension) is not stated it is assumed to be a 'call by value' parameter. Otherwise the parameter is a 'call by reference' parameter and (dimension) states the dimension of the parameter. 0 = simple variable, 1 = vector, and so on. Also 'ANYDIM' may be stated for (dimension). This will mean that the actual parameter may have any dimension and that it will be a 'call by reference' parameter. Then it is up to the extension to determine which dimension the parameter has and to handle it accordingly.

(type) states the type of the parameter which may be 'INT', 'REAL' or 'STR' (for integer, real or string respectively).

'ANYTYPE' or 'INTREAL' may be stated: 'ANYTYPE' means that the actual parameter may have any type ('INT', 'REAL', 'STR'), and 'INTREAL' means that the actual parameter may be of either 'INT' or 'REAL' type. The special rules for these two cases (regarding the way they are transferred to the extension) are described below.

For operators (interface) is either

```
OPERATOR (return type),(left operand type),
         (right operand type),(priority)
```

or

```
OPERATOR (return type),(operand type),(priority)
```

the former is used for dyadic operators and the latter for monadic operators. As with functions (return type) is either 'INT', 'REAL' or 'STR'.

(left operand type), (right operand type) and (operand type) may be 'INT', 'REAL', 'STR', 'ANYTYPE' or 'INTREAL' used as described for functions and statements above.

Parameters to operators can only be 'call by value' parameters.

(priority) states the priority of the evaluation of the operator in relation to other operators, including the standard operators.

(priority) is stated as the name of a standard operator, using the name used for a call to the mathematical package. The operator will then have the same priority as the standard operator. The following names may be used:

```
POWER
TIMES, SLASH, DIV, MOD,
PLUS, MINUS, CHS,
LEQ, LSS, GEQ, GTR, EQL, NEQ, IN,
B.NOT,
B.AND,
B.OR
```

After the line containing 'ENDALLEXT...' comes the code for the specified extensions. Each routine starts with a label which is <name> (or <local name>) of the corresponding 'EXTENSION' line.

When the routine starts, all registers used by COMAL-80 are saved and may be used. The parameters are pushed on the stack referenced by IX. This stack grows downwards like the SP-stack with the last parameter on the top.

For 'call by reference' parameters, a 'reference element' is placed on the stack (described below). Note that if the formal parameter is of the type 'ANYTYPE' or 'REALINT', the extension must find the type of the actual parameter and act accordingly. The type is determined by the reference element as described below.

For 'call by value' parameters, the value is on the stack. The way in which the value is structured is described below. If the formal parameter is of type 'ANYTYPE' or 'REALINT', an extra byte is pushed on top of the value and this byte states the type of the value in the following way: For 'ANYTYPE' the byte may be of the value 'INT', 'REAL' or 'STR', corresponding to the type of the actual parameter. For 'INTREAL' an actual parameter of type 'REAL' will always be converted (rounded if necessary) to 'INT' before the extension is called but the extra byte will state whether the conversion created an overflow (if the number was too large). In that case the byte will not be 0, otherwise it will be equal to 0. If the actual parameter is of type 'INT' the byte will be equal to 0. In the case of an overflow, the integer will still be in the stack (just below the byte) although its value will be of no interest.

The A and B registers contain, respectively, the version number of the running COMAL-80 version and the sub-version number. For version 2.0 the sub-version number is 0 and the version number is 8+DB+OV for which DB=0 for the 7-digit version and DB=2 for the 13-digit version, while OV=0 for the non-overlaid version and OV=1 for the overlay version.

Before the routine returns, all parameters must be popped from the IX-stack. If the routine corresponds to a function or an operator, the return value must be pushed onto the IX-stack before the return. However, if the return takes place while the C-bit is set (see below), it is not necessary to pop the parameters or push any return value. The return is made using 'RET'. Nothing must be done to change the size of the SP-stack (which may be used as there is about 100 bytes of free space).

Depending on the contents of the AF-register, COMAL-80 may give an error message on returning from the routine. Both fatal and non-fatal error messages are possible. If the Z-bit has been set, no error message will appear, otherwise the error message corresponding to the number in the A-register will be given, unless the number is above 150 in which case the error number is increased by 50. The error message will be fatal if the C-bit is set; otherwise it will be non-fatal, which means that it will not appear when a 'TRAP ERR-' statement has been included; the error number may be returned by using 'ERR()'.

STRUCTURE OF PARAMETERS

This section describes how parameters (values and reference elements) are represented in COMAL-80:

Integers: The bit-pattern 8000H means 'undefined' and it will never be supplied as a parameter. It should not be returned or assigned to any parameters. If a 'call by reference' parameter holds this value it means that the variable never has been assigned any value. This must be checked before the value is used. This is done automatically by the function 'LDVAL' of the mathematical package.

Real numbers use 4 bytes in the 7-digit version and 8 bytes in the 13-digit version. If the first byte is equal to 80H and the last byte to 00H, this means 'undefined' (see 'integers').

Strings consist of two parts: The first 2 bytes indicate the length of the string, the remaining bytes are a character-by-character representation of the string.

Reference elements denote variables of all kinds, both simple variables including string variables, as well as arrays and substrings. The format for reference elements for substrings is special and will be described separately; all other reference elements use 4 bytes and consist of two pointers: a pointer to a description of the variable, and a pointer to the data area of the variable.

The description of a variable is of varying size and grows towards lower addresses (NB!). For simple variables it consists of a byte carrying the value '-INT', '-REAL', or '-STR' corresponding the three types. If it is '-STR' this byte is followed by the maximum length of the string variable (an integer of 2 bytes).

For arrays a number of index-fields is first stated corresponding to the number of indices. Each index-field contains the following:

number of index-fields following	(1 byte)
lower limit for this index	(integer 2 bytes)
upper limit for this index	(integer 2 bytes)
size of the data of each sub-array or element	(integer 2 bytes)

Example:
The description of a variable for an array dimensioned by

```
'DIM A$(-2:5, 1:10) OF 20'
```

will be:

```
1       (1 byte)
-2      (2 bytes)
5       (2 bytes)
220     (2 bytes)

0       (1 byte)
1       (2 bytes)
10      (2 bytes)
22      (2 bytes)  (2 for actual length + 20 for the
                    characters)

-STR    (1 byte)
20      (2 bytes)
```

Reference elements for substrings also start with a pointer to a variable description which in this case is one single byte of either value '-STR-1' or '-STR-2'.

Then follows a pointer to the data-area for the string variable of which the substring is a part. Then follows

```
the maximum (dimensioned) length for the specific
    string variable,
2nd index (to-value), and
1st index (from-value)
```

which are used to specify the substring.

Examples of 2nd and 1st index are:

```
For 'A$(8:17)'
        2nd index = 17
        1st index = 8

For 'A$(10)'
        2nd index = 10
        1st index = 10
```

All four values above are integers (2 bytes).

The data-area for a simple variable is structured as for values. The data-area for an array is a continuous row of simple data-areas - one for each element. For string arrays, space is allocated for the maximum possible length of each element.

MATHEMATICAL PACKAGE

Within 'EXTDEFS.MAC' the necessary information for use of the mathematical package is defined. The mathematical package can be called using the macro

```
EXPR
```

End the call by using

```
EXPREND
```

The commands are executed one by one in the order in which they are written. Mostly, they take their arguments from the IX-stack and store their results on this stack. The commands may be considered as a program for a stack-oriented computer. In reality the commands are macros defined in 'EXTDEFS.MAC'. Each macro expands to one or more bytes (with parameters) and calls the mathematical package which interprets these parameters and operates on the IX-stack.

The following commands may all be used:

Corresponding to the standard functions of COMAL-80, the following commands may be used. All commands require arguments of a certain type and return a result of a certain type. The argument must be integer when 'INTREAL' is stated, but it will be accepted if it is the result of a call of 'REALINT' immediately before the command.

Name	Type of Argument(s)	Type of Result	
ATN	REAL	REAL	
COS	REAL	REAL	
SIN	REAL	REAL	
TAN	REAL	REAL	
LOG	REAL	REAL	
EXP	REAL	REAL	
SQR	REAL	REAL	
ESC	-	INT	
ERR	-	INT	
EOD	-	INT	
EOF	REALINT	INT	
LEN	ref.elemt.	INT	(ref.elemt. must be string)
ORD	STR	INT	
IVAL	STR	INT	
VAL	STR	REAL	
INT	REAL	REAL	
FRAC	REAL	REAL	
TRUNC	REAL	INT	
ROUND	REAL	INT	
POS	STR, STR	INT	
BVAL	STR	INT	
CHR	REALINT	STR	
STR	REAL	STR	(both correspond to STR$ but taken
I.STR	INT	STR	of a real number or an integer, respectively)

ERRTEXT	REALINT	STR	
SGN	REAL	INT	(both correspond to SGN but taken
I.SGN	INT	INT	of a real number or an integer, respectively)
ABS	REAL	REAL	(both correspond to ABS but taken
I.ABS	INT	INT	of a real number or an integer, respectively)
RND0	-	REAL	(RND without parameters)
RND2	REALINT, REALINT	INT	(RND with two parameters)
SPC	REALINT	STR	
PEEK	REAL	INT	
INP	REALINT	INT	
VARPTR	ref.elemt.	REAL	
FREEST	-	INT	(FREESTORE)

CONVERSIONS:

CONV converts the topmost stack element from 'INT' to 'REAL'.

CONV1 converts the topmost minus one stack element from 'INT' to 'REAL'. The top element is assumed to be of 'REAL' type.

REALINT converts the topmost stack element from 'REAL' to 'INT' including rounding if necessary. If the number is outside the integer area, the topmost stack element will be undefined. However, a special overflow-flag is set for suitable reaction by the standard functions for which 'REALINT' is stated above.

RLBL converts the topmost stack element from 'REAL' to 'INT' in such a way that the result will be 1 if the number is not 0, otherwise it will be 0.
This is used in connection with boolean operatos.

RLBL1 converts the topmost minus one stack element from 'REAL' to 'INT' in the same way as done by 'RLBL'. The top element is assumed to be an 'INT' type.

In this connection the standard operators of COMAL-80 have the following names:

Name	COMAL-80 Name	Type of the Argument(s)	Type of the Result
CHS	- (monadic)	REAL	REAL
POWER	^	)	
TIMES	*	)	
SLASH	/	)	
DIV	DIV	)-) REAL, REAL	REAL
MOD	MOD	)	
PLUS	+	)	
MINUS	- (dydadic)	)	

LEQ	<=	)	
LSS	<	)	
GEQ	>=	)-> REAL, REAL	INT
GTR	>	)	(value 0 or 1)
EQL	=	)	
NEQ	<>	)	
I.CHS	-	INT	INT
I.TIMES	*	)	
I.DIV	DIV	)	
I.MOD	MOD	)->INT, INT	INT
I.PLUS	+	)	
I.MINUS	-	)	
I.LEQ	<=	)	
I.LSS	<	)	
I.GEQ	>=	)->INT, INT	INT
I.GTR	>	)	(value 0 or 1)
I.EQL	=	)	
I.NEQ	<>	)	
S.PLUS	+	STR, STR	STR
S.LEQ	<=	)	
S.LSS	<	)	
S.GEQ	>=	)->STR, STR	INT
S.GTR	>	)	(value 0 or 1)
S.EQL	=	)	
S.NEQ	<>	)	
IN	IN	STR, STR	)
B.AND	AND	INT, INT	)->INT
B.OR	OR	INT, INT	) (value 0 or 1)
B.NOT	NOT	INT	)

OTHER COMMANDS:

INX expects a number followed by a reference element to an array or a string variable on top of the stack. Also, it performs indexing, i.e. it pops the stack and pushes a reference element for the chosen element of the stated array or the stated substring.
The number may be integer or real. If real, it is automatically converted to an integer with rounding. The type ('INT' or 'REAL') must be stated after 'INX', for example:

```
INX  REAL
```

LDVAL expects to find a reference element on top of the stack. It pops it and then pushes the value of the corresponding variable. 'LDVAL' may also be used after 'INX'. It checks that the variable is not 'undefined', i.e. that it does not carry the special value meaning that no value has been assigned to it.

STVAL expects a reference element followed by a value on the top of the stack. Both are popped and the value is stored in the variable given by the reference element. The type of value and that of the variable denoted by the reference element must conform. May be used after 'INX'.

LOAD expects an address (2 bytes) on the top of the stack. It pops the address and pushes the value/reference element at that address. The type of the value must be stated, unless it is stated as being a reference element. This is done in the same way as 'LOAD' and might look like this:

```
LOAD  INT
```

STORE expects an address (2 bytes) followed by a value or a reference element on the top of the stack. Both are popped and the value/reference element are stored at the given address. As with 'LOAD', the type of value must be stated, unless it is stated as being a reference element. Also, as with 'LOAD', an example might be:

```
STORE  REF
```

INTCON pushes an integer or an address (2 bytes) onto the stack. The integer/address must appear on the same line as 'INTCON' as:

```
INTCON  47
```

STRCON pushes a string onto the stack. The string value must appear on the same line as 'STRCON':

```
STRCON  'Strings appear in single quotaton marks'
```

UROUND expects a real number on the stack. The number is popped, converted to an unsigned integer between 0 and 65535, and pushed again.

SYSVAR pushes the value of one of the system variables as an integer. The system variable in question must be stated on the same line as 'SYSVAR', as in the example:

```
SYSVAR  PAGEWI
```

In connection with this the names of the system variables must be shortened according to the list below:

Name in COMAL-80	SYSVAR Name
ZONE	ZONE
INDENTION	INDENT
PAGEWIDTH	PAGEWI
PAGELENGTH	PAGELE
KEYWORDLOWER	KWLOWER
IDENTIFIERLOWER	IDLOWER

TRUE pushes a 1.

FALSE pushes a 0.

There is a possibility of errors occuring during the use of the mathematical package: overflow, index error, functions being given arguments, out of range, and so on. Here, there is a destinction between fatal and non-fatal errors. Errors which would be fatal during a normal COMAL-80 run will also be fatal in this connection. Errors which would not be fatal during a normal COMAL-80 run and consequently could be bypassed using 'TRAP ERR-' are also non-fatal here. The error number is stored and the execution of commands is resumed whenever a non-fatal error occurs. The error number will then be held in the A-register when returning to the extension after 'ENDEXPR', and the carry-bit will be 0, Z will be 1.

If a fatal error occurs the rest of the commands will be skipped, IX is set to the value held on entry to the mathematical package, and the system returns to the extension after 'ENDEXPR' with the error number in the A-register: carry = 1 and Z = 1. If no errors whatsoever occur A = 0 and Z = 0 on returning to the extension after 'ENDEXPR'. This means that it can be checked whether errors occur during the execution of a specific command by adding

```
        ENDEXPR
        JP      NZ,ERROR_HANDLING
        EXPR
```

after this command.

Note that these conventions covering the contents of A, CY, and Z correspond to the conventions for reporting errors back to the calling COMAL-80 program.

FORMAT FOR '.EXT' FILES

The file being loaded by COMAL-80 when using the 'EXTENSION' command must have a specific format. This is a very simple relocatable format. The assembler program, the form of which is described in the first chapter, can be translated by the Microsoft M80 Macro-assembler to one relocatable format. Other assemblers use other relocatable formats (and some of these even require that the definitions within 'EXTDEFS.MAC' are re-written). COMAL-80 defines its own relocatable format as follows:

The file starts with an integer of 2 bytes informing how much code is contained in the file (in bytes). This is not the same as the length of the file, as the file contains the code as well as information as to which parts of the code is to be relocated.

The rest of the file consists of a number of blocks with the same format. Each block consists of 9 bytes of which 8 contain code and the 9th holds information on the relocation of the 8 bytes. If the number with which the file starts is L, there will be (L + 7) DIV 8 blocks. If L is not a multiple of 8 there will be space left over in the last block. The contents of the remaining space is unimportant.

The code contained in the blocks represents one long row of code-bytes. Consequently, the code contained in two following blocks will follow right after each other when the file is read by COMAL-80. The byte carrying the information on relocation contains one bit for each of the 8 code-bytes so that bit 0 coresponds to the first byte and so on up to bit 7 which corresponds the 8th and last code-byte. If one of these bits is 1, it means that the address residing in that particular byte and the one preceeding are to be relocated. The two bytes are not to be relocated if the bit is 0.

The 'CONVERT' program can change a relocatable file in the Microsoft format into the corresponding file in the COMAL-80 format. If an assembler with another format is used, the user must write a program to change the relocatable file into COMAL-80 format.

All the COMAL-80 commands, statements and functions are described in this chapter. Each description is formatted as follows:

Type:	States whether a command, statement or function.
Purpose:	States what the instruction is used for.
Syntax:	Shows the correct syntax for the instruction. See below for syntax notation.
Execution:	Describes how the instruction is executed.
Example:	Shows sample programs or program segments that demonstrate the use of the instruction.
Comments:	Describes in detail how the instruction is used.

Syntax Notation.

Wherever the syntax of a statement, command or function is given, the following rules apply:

Items in capital letters must be entered as shown, using either upper or lower case letters.

Items in lower case letters and enclosed in angle brackets (< >) are inserted by the user.

Items in square brackets ([ ]) are optional.

All punctuation except angle brackets and square brackets (i.e. commas, parentheses, semicolons, colons, exclamation points, slashes, number signs, plus signs, minus signs and equal signs) must be included where shown.

All reserved words must be preceded by and/or followed by a space if this is necessary to avoid multiple interpretations.

Type:

Arithmetic function

Purpose:

To calculate the absolute value of an arithmetic expression

Syntax:

ABS(<expression>)

Execution:

Returns the absolute value of <expression>.

Example:

```
10 PRINT ABS(3*(-5))
```

Comments:

1. The result will be of the same type (real or integer) as the expression.

Type:

Logical operator

Purpose:

To perform the logical 'AND' between 2 expressions.

Syntax:

(expression1) AND (expression2)

Execution:

(expression1) is ANDed with (expression2).

Example:

```
10 INPUT A#
20 INPUT B#
30 IF A#=5 AND B#=7 THEN
40  PRINT "THE PRODUCT IS  35"
50 ELSE
60  PRINT "THE PRODUCT MAY NOT BE 35"
70 ENDIF
```

Comments:

1. This operator uses the truth table:

(expression1)	(expression2)	result
true	true	true
true	false	false
false	true	false
false	false	false

Type:

Arithmetic function

Purpose:

Returns the arctangent of an arithmetic expression.

Syntax:

ATN((expression))

Execution:

Returns the arctangent of (expression) in radians.

Example:

```
10 INPUT A
20 PRINT ATN(A)
```

Comments:

1. The result will always be real (whether (expression) is real or integer) and will lie between -pi/2 and pi/2.

Type:

Command

Purpose:

To generate a new line number automatically after each 'RETURN'.

Syntax:

AUTO [(start)[,(step)]]

Execution:

Following each 'RETURN' a new line number is calculated using the last line number used (or the value entered as (start)) plus the value of (step). The new number is placed in the input buffer and displayed on the screen.
The cursor is set to column 6 plus the current indent (which is program dependent) ready for a new input line.

Examples:

```
AUTO
AUTO 15
AUTO 10,5
```

Comments:

1. If the (start) value is omitted, default 10 is used.

2. If the (step) value is omitted, default 10 is used.

3. If an existing line number is generated, the new line replaces the former one.

4. The automatic generation of line numbers can be interrupted at any time by pressing the 'ESC' key.
   The line in which this is done will not be stored.

Type:

String function

Purpose:

Converts an arithmetic expression to binary representation.

Syntax:

BSTR$(<expression>)

Execution:

<expression> is calculated and rounded if necessary. The value is then converted to an 8 character binary text string.

Example:

```
10 DIM A$ OF 8
20 INPUT B
30 A$:=BSTR$(B)
40 PRINT A$
```

Comments:

1. <expression> must evaluate to a value between 0 and 255.

Type:

Arithmetic function

Purpose:

To convert a binary number from a string to an integer value.

Syntax:

BVAL ((string expression))

Execution:

A binary number contained in a string of 8 characters is converted to its integer form.

Example:

```
10 DIM A$ OF 8
20 DIM B$ OF 8
30 INPUT "WRITE A BINARY VALUE: ": A$
40 B$:="10101100"
50 PRINT BVAL(A$)
60 C#:=BVAL(B$)
70 PRINT C#
```

Comments:

1. If the string contains more than or less than 8 digits, or if it contains anything other than binary digits (0 and 1) program execution will be stopped with an error message.

Type:

Statement, command

Purpose:

To call a Z-80 machine code routine from COMAL-80.

Syntax:

CALL ⟨expression⟩

Execution:

⟨expression⟩ is calculated and rounded if necessary. The CPU then stores all its registers and executes a machine code routine starting at the specified address.

Examples:

```
    CALL 256
240 CALL 53248
```

Comments:

1. For further details on the Z-80 microprocessor and its assembler codes please refer to the manufacturers' manuals.

2. The user may use the CPU registers, however, the stack pointer must be re-established prior to returning to COMAL-80.

3. COMAL-80 does not utilize the interrupt facilities of the CPU. Consequently, the user may do this after returning to COMAL-80.

4. End the machine code with a 'RET' command to return to COMAL-80.

Type:

Statement

Purpose:

The case structure is used to select the program section to be executed according to the value of an expression.

Syntax:

```
CASE <expression> OF
WHEN <list of values>
.
.
WHEN <list of values>
.
.
WHEN <list of values>
.
[OTHERWISE
.
.]
ENDCASE
```

Execution:

The <expression> is evaluated and the 'WHEN' statements are checked one by one to find whether one of the list of values matches the calculated value.
When a match is found the lines from the 'WHEN' statement in which it is found, up to the next corresponding 'WHEN', 'OTHERWISE', or 'ENDCASE' statement, are executed, after which program execution continues after the 'ENDCASE' statement (provided that none of the executed lines have transferred the execution to yet another part of the program).
If none of the values fit the value of <expression> the lines following 'OTHERWISE' will be executed.
If 'OTHERWISE' is omitted, program execution stops with an error message if no match is found.

Example:

```
 10 DIM A$ OF 1
 20 INPUT "PRESS THE 'A' OR THE 'B' KEY":A$
 30 CASE A$ OF
 40 WHEN "A","a"
 50  PRINT "YOU HAVE PRESSED THE 'A' KEY"
 60 WHEN "B","b"
 70  PRINT "YOU HAVE PRESSED THE 'B' KEY"
 80 OTHERWISE
 90  GOTO 20
100 ENDCASE
```


Comments:

1. The expressions contained in the 'WHEN' statements must be of the same type as ⟨expression⟩ but integer expressions are allowed in the 'WHEN' statements if ⟨expression⟩ is of real type.

2. If several 'WHEN' statements correspond to ⟨expression⟩ only the first one will be executed.

Type:

Command

Purpose:

To display the catalog of a background storage device.

Syntax:

```
CAT [<file name1>[,<file name2>]]
CAT <file name2>
```

Execution:

The operating system of the computer is called and the contents of the file catalog are transferred to the specified <file name2>.

Examples:

```
CAT
CAT DK1:
CAT DK1:K
CAT DK1:,DK0:ABC.DEF
CAT *.CML,LP:
CAT DK1:C???????.*,LP:
CAT LP:
```

Comments:

1. <file name2> is the name of the file to which the catalog is output.

2. <file name1> specifies partly or wholly the name(s) of the catalog entries which are to be output. A partial specification may consist of a device name only (in which case the whole catalog of that device is output), or a partial file name, where the characters '*' and '?' are used following the CP/M protocol.

3. Omitting <file name2> displays the catalog on the terminal.

4. Omitting <file name1> displays the whole catalog of the current default device.

Type:

Statement

Purpose:

To write the catalog from a background storage device into a file.

Syntax:

CAT ⟨file name⟩, FILE ⟨file No.⟩

Execution:

The operating system of the computer is called, and information as to which device and which file names are to be written is passed to it. The catalog is written in ASCII format in the specified ⟨file No.⟩.

Examples:

```
100 CAT "DK1:", FILE 3
100 CAT "DK1:*.CML", FILE 2
```

Comments:

1. ⟨file name⟩ must be a string expression.

2. ⟨file name⟩ specifies the files required from a catalog.

3. ⟨file name⟩ specifies partly or wholly the name(s) of the catalog entries which are to be output. A partial specification may consist of a device name only (in which case the whole catalog of that device is output), or a partial file name, where the characters '*' AND '?' are used following the CP/M protocol.

4. If ⟨file name⟩ is an empty string, the whole catalog of the current default device will be displayed.

5. Before meeting the 'CAT' statement, a file carrying the stated ⟨file No.⟩ must be opened using the 'OPEN' statement.

6. The device on which the catalog is to be output must be specified in the 'OPEN' statement.

7. Following closing and re-opening, the created file may be read using the 'INPUT FILE' statement.

8. If a line printer with pagewidth = 0 or a diskfile is used for the printout, one file name is printed on each each line.

9. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.

Type:

Statement

Purpose:

To load and start execution of a program stored as a memory-image file on the background storage device.

Syntax:

CHAIN ⟨file name⟩ [,⟨list of variables⟩]

Execution:

The memory of the computer is cleared; the program ⟨file name⟩ is loaded and execution resumes from the lowest line number.

Example:

```
10 // MAIN PROGRAM
20 DIM PROGRAM$ OF 10
30 REPEAT
40  INPUT "WHICH PROGRAM IS WANTED? ": PROGRAM$
50 UNTIL PROGRAM$="LIST" OR PROGRAM$="UPDATE"
60 CHAIN PROGRAM$
```

Comments:

1. ⟨file name⟩ is a string expression.

2. If the 'CHAIN' statement includes a ⟨list of variables⟩ the new program section should have a 'RECEIVE ⟨list of variables⟩' statement.

3. This statement is used typically to organize a large program into smaller independent parts which may be loaded and executed according to user commands.

4. The program ⟨file name⟩ must be stored by the 'SAVE' command.

5. See also the 'RECEIVE' statement.

Type:

String function

Purpose:

To convert an arithmetic expression into a single-character string.

Syntax:

CHR$(⟨expression⟩)

Execution:

⟨expression⟩ is evaluated and rounded if necessary. The value is converted into a string consisting of a single character represented by that ASCII code.

Example:

```
10 INPUT A
20 PRINT CHR$(A)
```

Comments:

1. ⟨expression⟩ must be between 0 and 255.

Type:

Statement, command

Purpose:

To clear the screen and place the cursor in the top left corner.

Syntax:

CLEAR

Execution:

The screen is cleared and the cursor is placed in the top left corner.

Examples:

```
10 CLEAR
   CLEAR
```

Comments:

1. This statement/command affects the screen only.

Type:

Statement, command

Purpose:

To close one or more data files after use.

Syntax:

CLOSE [FILE ‹file No.›]

Execution:

The data file carrying the specified ‹file No.› is closed. ‹file No.›, which is an arithmetic expression, is evaluated and if necessary rounded before closing.

Examples:

```
200 CLOSE
390 CLOSE FILE 3
540 CLOSE FILE A*B
    CLOSE
```

Comments:

1. If 'FILE' and ‹file No.› are omitted, all open data files are closed.

2. When 'CLOSE' is executed, the stated connection between ‹file name› and ‹file No.› is detached and the file may be re-opened with the same or a new number.

3. Make sure that the 'CLOSE' statement/command is executed before program execution is finished to avoid leaving data in the system buffers.
   The 'RELEASE' command will indicate whether all files have been closed.

4. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.

Type:

Command

Purpose:

To resume program execution after a stop.

Syntax:

CON [<line No.>]

Execution:

Program execution is continued at <line No.> if specified, otherwise at the point at which it was stopped.

Examples:

```
CON
CON 220
```

Comments:

1. New values may be assigned to variables before resuming program execution.

2. Program execution may be resumed after a stop caused by a 'STOP' or an 'END' statement, after pressing the 'ESC' key and after a non-fatal error.

3. If the program stopped because of an error, program execution will be resumed starting with the statement in error. In all other cases program execution is started with the statement following the last statement executed.

4. If program editing has taken place, program execution cannot always be resumed.

5. If program execution is interrupted using the 'ESC' key while the computer is waiting in an 'INPUT' statement, a value will not be assigned to the variable in question. In this case program execution should be resumed by 'CON <line No.>' for the <line No.> displayed on the screen immediately after pressing the 'ESC' key.

Type:

Trigonometrical function

Purpose:

To calculate the cosine of an expression.

Syntax:

COS(⟨expression⟩)

Execution:

Cosine of ⟨expression⟩, for which ⟨expression⟩ is in radians, is calculated.

Example:

```
10 INPUT A
20 PRINT COS(A)
```

Comments:

1. ⟨expression⟩ may be an arithmetic expression of real or integer type. The result will always be real.

Type:

Statement, command

Purpose:

To place the cursor at a specified position on the screen.

Syntax:

CURSOR ⟨expression1⟩, ⟨expression2⟩

Execution:

⟨expression1⟩ and ⟨expression2⟩, both of which must be arithmetic expressions, are evaluated and rounded. The cursor is then moved to the column defined by ⟨expression1⟩ and to the line number defined by ⟨expresion2⟩.

Examples:

```
100 CURSOR 8, 12
220 CURSOR CHARACTER#,LINE#
300 CURSOR 3*2, 5+4
    CURSOR 10, 15
```

Comments:

1. ⟨expression1⟩ is counted from left to right and ⟨expression2⟩ is counted as positives from the top down. The top left corner has the coordinates 1,1.

Type:

Statement

Purpose:

To define constants in the form of a data list to be read by the 'READ' statement.

Syntax:

DATA ⟨constant1⟩,⟨constant2⟩,.....,⟨constantn⟩

Execution:

At the start of program execution, a search is made for 'DATA' statements and they are chained into a data list. During a run, an internal pointer is set to the next constant in the list.

Example:

```
10 DIM FIRST_NAME$ OF 10
20 DIM FAMILY_NAME$ OF 15
30 DATA "JOHN","DOE"
40 READ FIRST_NAME$
50 READ FAMILY_NAME$
60 PRINT FIRST_NAME$+" "+FAMILY_NAME$
70 DATA 35
80 READ AGE
90 PRINT AGE; "YEAR"
```

Comments:

1. 'DATA' statements are non-executable and are skipped during program execution.

2. Any number of 'DATA' statements may be placed anywhere in the program.

3. A 'DATA' statement may contain as many constants (separated by commas) as are allowed by the maximum length of an input line (159 characters).

4. The 'READ' statement reads the 'DATA' statements in line number order.

5. The types of constants may be mixed but must match those of the corresponding 'READ' statements otherwise execution results in an error message.
   Arithmetic expressions are not allowed in a 'DATA' statement, and string constants must be enclosed in double quotation marks.

6. The constants may be re-read, partly or wholly, by means of 'RESTORE', 'RESTORE ⟨line number⟩', or 'RESTORE ⟨name⟩' statements.

7. When the last constant is read the system function 'EOD()' will return the value of true (= 1).

Type:

Command

Purpose:

To delete one or more program lines.

Syntax:

```
DEL <start line>[,<end line>]
DEL ,<end line>
DEL <start line>,
```

Execution:

The specified line(s) is/are deleted from the program.

Examples:

```
DEL 25,100
DEL ,220
DEL 95,
DEL 40
```

Comments:

1. If <start line> only is specified, this line alone will be deleted.

2. If <start line> immediately followed by a comma is specified, this line and the rest of the program will be deleted.

3. If a comma followed by a line number only is specified, the program is deleted up to and including this line.

4. Specifying <start line> comma <end line> deletes the lines between the two inclusively.

Type:

Statement, command

Purpose:

To delete file(s) on the background storage device.

Syntax:

DELETE <file name>

Execution:

The operating system is called and information on the file(s) to be deleted is passed to it.

Examples:

```
100 DELETE "TEST.CML"
220 DELETE "DK1:DATA.DAT"
300 DELETE "DK0:D???????.*"
    DELETE PROGRAM.CML
    DELETE DK1:C*.CML
```

Comments:

1. In statements <file name> must be a string expression.

2. <file name> specifies partly or wholly the name(s) which is/are to be deleted. The characters '*' and/or '?' can be used following the CP/M protocol.

3. The whole file name, including any extension, must be specified.

4. If <filename> does not exist then an error message is given if 'DELETE' has been used as a command - but not if it has been used as a statement.

Type:

Statement

Purpose:

To allocate memory space for arrays and set to the index limits.

Syntax:

DIM (list of indexed variables)

Execution:

The necessary memory is calculated and allocated according to the type of variable.

Examples:

```
10 DIM MONKEY(5)
10 DIM NUMBER(7,3), COUNT(7)     // SEE NOTE 5
10 DIM CARS#(-5:15,3:8)
10 DIM A$(3:2), B(5)             // SEE NOTE 6
```

Comments:

1. Arrays must be dimensioned.

2. An array may have any number of dimensions limited only by the memory available and the maximum length of an input line (159 characters).

3. Each of the elements in (list of indexed variables) is specified using the syntax:
   (variable name)((list of index limits))
   where (variable name) optionally includes the declaration character '#'.
   The elements are separated using commas.
   (list of index limits) contains the lower and upper limits for each dimension following the syntax:
   [(lower limit):](upper limit)
   The dimensions are separated by commas.
   If no lower limit is given, a default of 1 is used.

4. The 'DIM' statement assigns the value 0 to each element.

5. Several variables can be dimensioned in the same line.

6. Arithmetic and string variables can be dimensioned on the same line.

Type:

Statement

Purpose:

To allocate memory space for strings and arrays of strings and set the index limits.

Syntax:

DIM (list of indexed variables)

Execution:

The necessary memory is allocated according to the dimensions and length of the variable.

Examples:

```
10 DIM A$ OF 80                          // SEE NOTE 9
10 DIM A$(3) OF 10                       // SEE NOTE 7
10 DIM B$(0:1,3) OF 25                   // SEE NOTE 8
10 DIM A$(3:2) OF 10, B$(5) OF 25        // SEE NOTE 5
10 DIM A$(5) OF 15, C(5)                 // SEE NOTE 6
```

Comments:

1. Arrays and string variables must always be dimensioned.

2. An array may have any number of dimensions limited only by the memory available and the maximum length of the input line (159 characters).

3. Each of the elements in (list of indexed variables) is specified using the syntax:
   (variable name)[((list of index limits))] OF (length)
   where (variable name) includes the declaration character '$'.
   The elements are separated using commas.
   (list of index limits) contains, for each dimension of an array, upper and lower limits for that dimension following the syntax:
   [(lower limit):](upper limit)
   The dimensions are separated by commas.
   If no lower limit is given a default value of 1 is used.
   (length) indicates the maximum length of the string variable or of each of the elements in the string array.
   The actual value of a string variable/element may vary from zero characters (the empty string) up to and including the stated (length).

4. The 'DIM' statement assigns the value "" (empty string) to each element.

5. Several variables can be dimensioned in the same line.

6. Arithmetic and string variables can be dimensioned in the same line.

7. This array will contain the elements A$(1), A$(2) and A$(3) each having a maximum length of 10 characters.

8. This array will contain the elements B$(0,1), B$(0,2), B$(0,3), B$(1,1), B$(1,2) and B$(1,3) each having a maximum length of 25 characters.

9. A string variable need not be an array.

Type:

Arithmetic operator

Purpose:

To carry out an integer division between two arithmetic expressions.

Syntax:

⟨expression1⟩ DIV ⟨expression2⟩

Execution:

⟨expression1⟩ is divided by ⟨expression2⟩ and the result is rounded to an integer value.

Examples:

```
100 A#:=B DIV C
100 NUMBER:=17 DIV NUM
```

Comments:

1. The result N is defined by the integer value of N which makes the expression
   ⟨expression1⟩ - N * ⟨expression2⟩
   assume its lowest possible non-negative value.

2. The type of the result depends upon the type of ⟨expression1⟩ and ⟨expression2⟩ in the following way:

⟨expression1⟩	DIV	⟨expression2⟩	result
real		real	real
real		int	real
int		real	real
int		int	int

3. Also see the 'MOD' operator.

Type:

Command

Purpose:

To simplify correction of a program held in working memory.

Syntax:

```
EDIT [(start)][,(end)]
EDIT [(start),]
```

Execution:

The specified program area is called from working storage and displayed on the screen line by line. The cursor is placed immediately after the last character and can be moved backwards and forwards on the line using the cursor control keys. Place the cursor over the character to be corrected, key in the correction and the cursor will move one position to the right.
When the corrections are complete, press 'RETURN'. The line undergoes the syntax control and when accepted it is stored. The next line is displayed and the sequence repeats until (end) is reached.

Examples:

```
EDIT
EDIT 100
EDIT 100,
EDIT ,100
EDIT 100,200
```

Comments:

1. If (start) is omitted, the editing starts at the first program line.

2. If (end) is omitted, the editing continues until the end of the program.

3. Omitting both limits, starts the editing at the first program line and continues to the end of the program (or until the 'ESC' key is pressed).

4. If only (start) is used, without a comma, editing will be restricted to the one line.

5. All the correction facilities described in INPUT EDITING in chapter 1 are available.

6. The line number itself may be edited causing the line to be placed in memory at the new line number. Any line already stored at that number will be deleted.
   The original line will not have been deleted from the program (use the 'DEL' command).

7. On pressing 'RETURN' the entire line is stored in memory regardless of the position of the cursor.

8. The edit command may be interrupted at any time by pressing the 'ESC' key. Changes in the line will be entered only after pressing 'RETURN'.

Type:

Statement

Purpose:

To stop the execution of a program.

Syntax:

END

Execution:

Program execution is terminated and the prompt character '*' is displayed to show that the COMAL-80 interpreter is ready to accept new input.

Example:

```
10 K:=0
20 K:+1
30 IF K)100 THEN END
40 PRINT K
50 GOTO 20
```

Comments:

1. The 'END' statement does not give any information as to where program execution was stopped (see 'STOP').

2. The use of the 'END' statement is optional, as COMAL-80 adds an invisible statement to the end of each program. When this statement is reached, the following message is displayed:

   Program execution finished

Type:

Command

Purpose:

To transfer a file from the background storage device to working memory in ASCII format.

Syntax:

ENTER (file name)

Execution:

The specified file is opened and transferred one character at a time.
Following each 'RETURN' the line is syntax-checked and the line, if accepted, is placed in the working memory. If an error occurs then the loading is halted temporarily, the line is displayed with an error message and the loading of the file continues. A line containing a syntax error will not be stored.

Examples:

```
ENTER DKO:PROGRAM
ENTER POLYNO
```

Comments:

1. Only files stored in ASCII format can be read by the 'ENTER' command. Thus files created by means of 'SAVE' cannot be read in this way. Use 'LOAD' instead.

2. The working memory is not cleared prior to the file being entered. However, new lines having line numbers which match existing lines will replace the old ones. This overwriting takes place on a line basis, with no consideration of the different lengths of lines, so that a short line can totally replace a long one. Provided that there are no overlapping line numbers this system may be used to combine two or more programs.
   Normally the working memory would be cleared by using the 'NEW' command before reading a file with the 'ENTER' command.

3. ASCII files may be read by all versions of COMAL-80 and this format is recommended for long-term storage of files.

Type:

System function

Purpose:

To determine whether all data from the 'DATA' statements in the program has been read.

Syntax:

EOD()

Execution:

EOD() returns a value of false (= 0) as long as there is data in 'DATA' statements still to be read. Having read the last item of data, 'EOD()' will return the value of true (= 1). After executing a 'RESTORE' statement 'EOD()' will return the value of false (= 0).

Example:

```
10 WHILE NOT EOD() DO
20  READ A
30  PRINT A
40 ENDWHILE
50 DATA 55, 2, -15, 35
```

Comments:

1. During programming 'EOD' and 'EOD()' are interchangeable. In program listings 'EOD()' is used.

Type:

System function

Purpose:

To determine whether all data in a data file has been read.

Syntax:

EOF ((file No.))

Execution:

After execution of an 'OPEN FILE' statement or a 'READ' command, the corresponding 'EOF ((file No.))' will return the value false (= 0). After reading the last item in the file, it will return the value true (= 1).

Example:

```
10 OPEN FILE 0,"TEST",READ
20 REPEAT
30  READ FILE 0: A
40 UNTIL EOF(0)
```

Comments:

1. (file No.) is an arithmetic expression.

Type:

System function

Purpose:

To return the number of a non-fatal error encountered during program execution.

Syntax:

ERR()

Execution:

During normal program execution, any error will stop the program and create an error message. However, a number of errors can subsequently be bypassed.
In such cases program interruption may be avoided by the use of a 'TRAP ERR-' statement before the error arises. In this case, 'ERR()' will return a value equal to the error number of the last error and in all tests will be considered as true (because it is not 0). Program execution will then continue.

Example:

```
 10 INIT " "
 20 TRAP ERR-
 30 OPEN FILE 0,"DK1:DISK_5",READ
 40 TRAP ERR+
 50 IF NOT ERR() THEN
 60  CLOSE
 70  CHAIN "DK1:PART5"
 80 ELSE
 90  CLEAR
100  CURSOR 1,10
110  PRINT "WRONG DISK IN DRIVE 1"
120  STOP
130 ENDIF
```

Comments:

1. At the beginning of the execution of a program the value false (= 0) is returned if 'ERR()' is called.
   When a 'TRAP ERR-' statement has been executed, a non-fatal error will not stop program execution. The number of the error is returned by 'ERR()'. However, subsequent calls of 'ERR()' will return 0. This way only information on the last error that has occured since the last call of 'ERR()' will be retrieved. Since every value different from 0 is treated as true in tests, constructions like 'IF ERR() THEN...' will behave normally.
   The error numbers are described further in appendix C.

2. By executing a 'TRAP ERR+' statement, the system returns to normal error handling.

3. During programming 'ERR' and 'ERR()' are interchangeable. In program listings 'ERR()' is used.

Type:

String function

Purpose:

To give access to error descriptions in the COMAL-80 system

Syntax:

ERRTEXT$((expression))

Execution:

(expression) is evaluated and rounded if necessary. The corresponding error description is then returned.

Example:

```
10 FOR I=1 TO 295
20  PRINT ERRTEXT$(I)
30 NEXT I
```

Comments:

1. This function is only valid when error descriptions are not deleted at the start-up of COMAL-80. If they are deleted the function will return an empty string.

Type:

System function

Purpose:

To flag the use of the 'ESC' key.

Syntax:

ESC()

Execution:

During normal program execution a check is made to see whether the 'ESC' key has been pressed. If it has been pressed then program execution is stopped.
If a 'TRAP ESC-' statement has been executed, this function is blocked and 'ESC()' will instead return the value of true (= 1) when 'ESC' is pressed.

Example:

```
10 TRAP ESC-
20 REPEAT
30  PRINT "THE 'ESC' KEY HAS NOT BEEN PRESSED"
40 UNTIL ESC()
50 TRAP ESC+
60 PRINT "THE 'ESC' KEY HAS BEEN PRESSED"
```

Comments:

1. At the start of program execution 'ESC()' will return the value false (= 0). If a 'TRAP ESC-' statement is executed and the 'ESC' key pressed after this program execution continues but the first call of 'ESC()' will return the value true (= 1). Any subsequent calls will again return the value false (= 0).

2. The system returns to normal handling of the 'ESC' key after a 'TRAP ESC+' statement has been executed.

3. During programming 'ESC' and 'ESC()' are interchangeable. In program listings 'ESC()' is used.

Type:

Statement

Purpose:

To call a named sub-program and to return to the next line of the current program on completion.

Syntax:

EXEC ⟨procedure name⟩[(⟨actual parameter list⟩)]

Execution:

The procedure specified by ⟨procedure name⟩ is called, and ⟨actual parameter list⟩ replaces the formal parameter list in the procedure heading.
On reaching the 'ENDPROC' statement, program execution is resumed from the first executeable line following the 'EXEC' statement.

Examples:

```
100 EXEC TEST
100 EXEC FATAL_ERROR("ERROR IN X-PL/0-COMPILER")
100 EXEC ERROR(30)
100 EXEC ENTER_(CONSTANT#,LEV#,TX#,DX#)
```

Comments:

1. The number of actual parameters must be the same as the number of formal parameters in the 'PROC' statement. Each parameter must conform in dimension and type.

2. If a formal parameter is specified by 'REF', a variable (which may be indexed) must be inserted as an actual parameter.

3. If a formal parameter is not specified by 'REF' the actual parameter must be an expression of a corresponding type, a variable name alone will suffice.
Actual integer parameters may be inserted in a formal real parameter.

4. See the section 'PARAMETER SUBSTITUTION' in chapter 1 for more information.

Type:

Arithmetic function

Purpose:

Returns e to the power of an arithmetic expression.

Syntax:

EXP(⟨expression⟩)

Execution:

The base of the natural logarithm e (2.718282) is raised to the power specified by ⟨expression⟩.

Example:

```
10 INPUT A
20 PRINT EXP(A)
```

Comments:

1. ⟨expression⟩ is a real or integer arithmetic expression. The result will always be real.

2. The value of ⟨expression⟩ must be less than or equal to 88.02968 when using the COMAL-80 7-digit version and 292.4283068102 when using 13-digit version. If these are exceded COMAL-80 stops program execution and displays an error message.

Type:

Command

Purpose:

To add user-defineable statements, functions, and operators to COMAL-80.

Syntax:

EXTENSION <file name>

Execution:

<file name> is opened and transferred to the memory. The identifiers specified in this file are then linked to COMAL-80 and become reserved words.

Example:

```
EXTENSION GRAPHPAC
```

Comments:

1. This command is only allowed when there is no program in memory.

2. See the section 'EXTENSION' in chapter 1 and appendix D for further information.

Type:

System constant

Purpose:

To assign a boolean variable the value of false.

Syntax:

FALSE

Execution:

Returns the value 0.

Example:

```
 10 // PRIME
 20 //
 30 DIM FLAGS#(0:8190)
 40 SIZE1:=8190
 50 //
 60 COUNT:=0
 70 MAT FLAGS#:=TRUE
 80 //
 90 FOR I:=0 TO SIZE1 DO
100  IF FLAGS#(I) THEN
110   PRIME:=I+I+3
120   K:=I+PRIME
130   WHILE K(=SIZE1 DO
140    FLAGS#(K):=FALSE
150    K:+PRIME
160   ENDWHILE
170   COUNT:+1
180  ENDIF
190 NEXT I
200 PRINT "TOTAL NUMBER OF PRIMES: ",COUNT
```

Type:

Statement

Purpose:

To delimit a program section and define the number of times it is to be executed.

Syntax:

```
FOR (variable) := (start) TO (end) [STEP (step)]
.
.
.
.
NEXT (variable)
```

Execution:

On meeting the 'FOR' statement, (variable):=(start) is assigned and the truth of:

((end)-(variable))*SGN ((step)))= 0

is tested. If this is false, the 'FOR...NEXT' structure, including this program section, is bypassed and execution continues from the first executable line following the 'NEXT' statement.

If true the program continues through the program section until it meets a 'NEXT' statement; it then jumps back to the line following 'FOR' adding (step) to (variable) and checks the truth again using the new value of (variable). This is repeated until the test returns false.

Example:

```
10 FOR I=1 TO 100 STEP 5
20  PRINT I, " ",
30 NEXT I
40 STOP
```

Comments:

1. If 'STEP (step)' is omitted the (step) value defaults to 1.

2. If 'DOWNTO' is used instead of 'TO', the negative value of (step) is used as the step value.

3. Following a 'FOR...NEXT' execution, (variable) takes the value not fulfilling the above test.

4. Up to five 'FOR...NEXT' statements may be nested, each of them having their separate (variable).
   Each subroutine level is assigned a 'FOR...NEXT' depth of five, giving the option of any depth through the 'GOSUB' statement or by use of procedures.

5. Each 'NEXT' statement may contain one only ⟨variable⟩, which must be the same one as stated in the corresponding 'FOR' statement.

6. It is possible to interrupt a 'FOR...NEXT' sequence by using 'GOTO'.

7. The start value of the ⟨variable⟩ is assigned before ⟨end⟩.
   Consequently program structures of the type:

```
10 J:= X
20 FOR J:=1 TO J+X
30  PRINT J
40 NEXT J
```

   will be executed X+1 times.

8. Only one 'NEXT' statement may be assigned for each 'FOR' statement.

9. During programming ':=' and '=' are interchangeable. In program listings ':=' is used.

10. ⟨variable⟩ must be an arithmetic variable.

Type:

Arithmetic function

Purpose:

To extract the decimal part of a real number.

Syntax:

FRAC(⟨expression⟩)

Execution:

The result is calculated according to the expression:

⟨expression⟩-INT(⟨expression⟩)

Example:

```
10 INPUT A
20 PRINT FRAC(A)
30 PRINT FRAC(5.72)
40 PRINT FRAC(-5.72)
```

Comments:

1. ⟨expression⟩ must be arithmetic and real. The result will be real.

2. If ⟨expression⟩ is positive the result is calculated by cancelling the digits in front of the decimal point. If ⟨expression⟩ is negative the result is 1 minus the decimal part of ⟨expression⟩.

Type:

System function

Purpose:

To return the number of bytes of free memory space.

Syntax:

FREESTORE()

Execution:

The available free space is calculated based on the current use of the memory.

Example:

```
10 PRINT FREESTORE()
```

Comments:

1. During programming 'FREESTORE' and 'FREESTORE()' are interchangeable. In program listings 'FREESTORE()' is used.

Type:

Statement

Purpose:

To define and name a user-defined function.

Syntax:

```
FUNC (name)[(formal parameter list)] [CLOSED]
.
.
ENDFUNC (name)
```

Execution:

When finding a 'FUNC' statement during program execution, COMAL-80 skips this part of the program up to and including the corresponding 'ENDFUNC' statement and execution is resumed from the next line.
When the function is called by its name (optionally followed by a parameter list) in an expression, the function is calculated and the value is inserted in the expression and used in the subsequent calculation.

Examples:

```
10 FUNC X_Y_POWER(X,Y)
20  RETURN X^3/Y^2
30 ENDFUNC X_Y_POWER
40 I:=2
50 J:=3
60 OLE:=X_Y_POWER(I,J)
70 PRINT OLE
```

```
10 X:=2
20 Y:=3
30 FUNC X_Y_POWER CLOSED
40  IMPORT X,Y
50  RETURN X^3/Y^2
60 ENDFUNC X_Y_POWER
70 OLE:=X_Y_POWER()
80 PRINT OLE
```

Comments:

1. The 'FUNC' statement may not be used within the following statements:
   - Conditional statements
   - Repeating statements
   - Other procedure or function declarations

2. (name) must be a legal variable name.

3. A function may call other functions, and may call itself (recursion). A closed function can only call a closed function or procedure.

4. (formal parameter list) contains the names of the formal parameters which will receive values from the actual parameters in the function call when called.

5. The changes happening to a parameter in a function are local unless 'REF' has been used to indicate that the changes are to affect the actual parameter.

6. 'REF' may be stated for simple arithmetic or string variables, and must be stated for all array variables.

7. A function type may be either real, integer or string.

8. Array variables must be followed by a dimension definition consisting of commas in parentheses corresponding to the number of dimensions -1, i.e. for 3-dimensional arrays the parenthesis contains two commas, while a vector would be followed by an empty parenthesis.

9. If the function is declared 'CLOSED', all variable names are local and may be used for other purposes outside the function. This may be declared invalid for one or more variables by use of the 'IMPORT' statement.

10. The 'INPUT' and 'PRINT USING' statements are not allowed in functions.

11. If the program section between 'FUNC' and 'ENDFUNC' contains statements on multiple lines these must all be contained in the program section.

12. The function value is returned from the function by the 'RETURN' statement. Otherwise the value of the function is undefined.

Type:

Statement, command

Purpose:

Returns the current background storage device.

Syntax:

GETUNIT [<variable>]

Execution:

The name of the current default device is assigned to <variable> in the form of a 3-character code, two letters and one digit followed by a colon.

Examples:

```
100 GETUNIT DISK$
    GETUNIT
```

Comments:

1. When using 'GETUNIT' as a command the <variable> must be omitted, and the result will be displayed on the terminal.
   In statements the <variable> must be specified.

2. The two letters indicate the type of device; 'DK' means floppy disk. The digit indicates the unit number.

3. <variable> is a string variable.

Type:

Statement

Purpose:

To call a subroutine at a different part of the program and then return to the line following the call.

Syntax:

```
                    GOSUB (line number)
                    .
                    .
(line number)       .
                    .
                    RETURN
```

Execution:

On meeting a 'GOSUB' statement, program execution continues from (line number) until it reaches the 'RETURN' statement when program execution is resumed from the line following the 'GOSUB' statement.

Example:

```
10 PRINT "START IN THE MAIN PROGRAM"
20 GOSUB 50
30 PRINT "BACK IN THE MAIN PROGRAM"
40 STOP
50 PRINT "THIS IS THE SUBROUTINE"
60 RETURN
```

Comments:

1. A subroutine may be called any number of times.

2. Subroutines may be called from other subroutines, and such nestings are limited only by the available memory.

3. Following the 'RETURN' statement program execution is resumed from the line immediately following the 'GOSUB' executed.

4. A subroutine may include more than one 'RETURN' statement.

5. Subroutines may be placed anywhere in the main program, but clear separation from the main program listing is recommended.

6. To prevent any inadvertant execution of a subroutine it is a good idea to put a 'STOP', 'GOTO', or an 'END' statement immediately before the subroutine.

7. NOTE. The keyword 'RETURN' may also be used in procedures and functions. This is described in details on page 2-098.

Type:

Statement

Purpose:

To interrupt normal sequential program execution and continue from the stated line.

Syntax:

GOTO <line number>
GOTO <name>

Execution:

The execution continues at the stated line or, if this cannot be executed, from the first following executable line.

Examples:

```
10 PRINT "JO",
20 GOTO 40
30 STOP
40 PRINT "HN"
50 GOTO 30
```

```
10 PRINT "JO",
20 GOTO REST
30 LABEL FINISH
40 STOP
50 LABEL REST
60 PRINT "HN"
70 GOTO FINISH
```

Comments:

1. Statements such as 'LABEL' and 'REM' are not executable.

Type:

System variable

Purpose:

To specify whether identifiers in program listings are to appear in upper or lower case letters.

Syntax:

IDENTIFIERLOWER

Execution:

The value of the system variable 'IDENTIFIERLOWER' controls the format of identifiers in program listings.

Examples:

```
100 IDENTIFIERLOWER:=0
100 IDENTIFIERLOWER:=A
100 IDENTIFIERLOWER:=TRUE
100 PRINT IDENTIFIERLOWER
    IDENTIFIERLOWER:=1
```

Comments:

1. On loading COMAL-80 'IDENTIFIERLOWER' is assigned the value of 0. This value can only be changed by an assignment to 'IDENTIFIERLOWER'.

2. The value assigned must be 0 or 1. Assigned values are rounded if necessary.

3. If the value of 'IDENTIFIERLOWER' is equal to 0 all identifiers will be listed in upper case. Otherwise they will be listed in lower case.

4. This keyword can be used as operand or be assigned to. When used as operand it is of integer type.

5. The 'NEW' command does not change the value of the system variable 'IDENTIFIERLOWER'.

Type:

Statement

Purpose:

To execute or skip a statement depending on a logical expression being true or false.

Syntax:

IF <logical expression> [THEN] <statement>

Execution:

Only when <logical expression> is true (<> 0), is <statement> executed.

Example:

```
10 INPUT "PRINT A NUMBER: ": A
20 IF A THEN PRINT "A <> 0"
30 IF A<0 THEN PRINT "A<0"
40 IF A=0 THEN PRINT "A=0"
50 IF A=1 THEN PRINT "A=1"
60 IF A=2 THEN PRINT "A=2"
70 IF A>2 THEN PRINT "A>2"
```

Comments:

1. The following statements may be used after an 'IF... THEN' statement:
   CALL, CAT, CHAIN, CLEAR, CLOSE, CURSOR, DELETE, END, EXEC, EXIT, GETUNIT, GOSUB, GOTO, INIT, INPUT, LET, LOGOFF, LOGON, MAT, ON, OPEN, OUT, PAGE, POKE, PRINT, QUIT, RANDOM, READ, RECEIVE, RELEASE, RENAME, RESTORE, RETURN, SELECT, STOP, TRAP, UNIT, WRITE, and statements defined as 'EXTENSIONS'.
   A new 'IF...THEN' statement is also allowed.

2. During programming 'THEN' may be omitted as COMAL-80 automatically adds it to program listings.

Type:

Statement

Purpose:

To execute a program section if a logical expression is true. Otherwise the section is skipped.

Syntax:

```
IF (logical expression) [THEN]
.
.
.
ENDIF
```

Execution:

If the (logical expression) is true (<> 0) the program section within 'IF...ENDIF' is executed. If the (logical expression) is false (= 0) the program is resumed from the first executable line following the 'ENDIF' statement.

Example:

```
10 IF MEMBER#<1 OR MEMBER#>31 THEN
20  EXEC FATALERROR("ERROR IN X-PL/0-COMPILER")
30 ENDIF
```

Comments:

1. During programming 'THEN' may be omitted, as COMAL-80 automatically adds it to program listings.

Type:

Statement

Purpose:

To execute one of two program sections depending on a logical expression being true or false.

Syntax:

```
IF <logical expression> [THEN]
.
.
.
ELSE
.
.
.
ENDIF
```

Execution:

If the <logical expression> is true (<> 0) the program section surrounded by 'IF...ELSE' is executed. If the <logical expression> is false (= 0) the program section surrounded by 'ELSE...ENDIF' is executed.

Example:

```
10 INPUT "GUESS A NUMBER BETWEEN 1 AND 5": A
20 B:=RND(1,5)
30 IF A=B THEN
40  PRINT "CORRECT"
50 ELSE
60  PRINT "WRONG. THE NUMBER WAS: "; B
70 ENDIF
80 STOP
```

Comments:

1. During programming 'THEN' may be omitted as COMAL-80 automatically adds it to program listings.

Type:

Statement

Purpose:

To execute one of several program sections depending on one of several logical expressions being true.

Syntax:

```
IF (logical expression 1) [THEN]
.
.
ELIF (logical expression 2) [THEN]
.
.
ELIF (logical expression n) [THEN]
.
.
[ELSE
.
.]
ENDIF
```

Execution:

Each (logical expression ..) is checked one by one. If one is true (<> 0) the following program section is executed until it meets the corresponding 'ELIF', 'ELSE', or 'ENDIF' statement. The program resumes from the first executable line following the 'ENDIF' statement.
When all (logical expressions) are false (= 0) the program section surrounded by 'ELSE...ENDIF' is executed and the program is resumed from the first executable line following the 'ENDIF' statement.

Example:

```
 10 INPUT "PRESS ONE OF THE DIGITS 1, 2, OR 3: ": A,
 20 IF A=1 THEN
 30  PRINT "THE DIGIT WAS 1"
 40 ELIF A=2 THEN
 50  PRINT "THE DIGIT WAS 2"
 60 ELIF A=3 THEN
 70  PRINT "THE DIGIT WAS 3"
 80 ELSE
 90  PRINT "I ASKED FOR ONE OF THE DIGITS 1, 2, OR 3!"
100 ENDIF
```

Comments:

1. If several (logical expressions) are true, only the first one is evaluated.

2. If there is no 'ELSE' statement, and if none of the (logical expressions) are true, program execution continues in the first line after 'ENDIF'.

3. During programming 'THEN' may be omitted, as COMAL-80 automatically adds it to program listings.

Type:

Statement

Purpose:

To make variables in the main program or another 'PROC' or 'FUNC' accessible within a 'PROC' or 'FUNC' structure.

Syntax:

IMPORT <list of variable names>

Execution:

The variables listed in <list of variable names> are made accessible within the 'PROC' or 'FUNC' structure containing the 'IMPORT' statement.

Example:

```
10 PROC ERROR(N#) CLOSED
20  IMPORT FATAL_ERROR:CC#, ERR_, ERRORS#
30  PRINT "*****"; SPC$(CC#-9); "^"; N#
40  ERR_:=INCLUDE(ERR_,N#+1); ERRORS#:+1
50 ENDPROC ERROR
```

Comments:

1. The variable names in <list of variable names> must be separated by commas. Array variable names should not be followed by any subscripts.

2. Each variable name in <list of variable names> may be preceded by a <closed area name> where <closed area name> is the name of the closed function or procedure from which the variable is taken.
   The <closed area name> must be the caller of the 'PROC/FUNC' containing the 'IMPORT' statement or it must be the caller of the caller, etc. The variable is taken from the most recent call of a <closed area name>.
   If <closed area name> is omitted, the variable is taken from the main program.

3. This statement may be used only within closed 'PROC' and 'FUNC' structures.

4. The execution of the 'IMPORT' statement does not affect the accessibility of the listed variables in any part of the program other than the 'PROC' or 'FUNC' structure containing the 'IMPORT' statement.

5. All operations allowed on the variables in the main program are also allowed within the 'PROC' or 'FUNC' structure containing the 'IMPORT' statement.

6. During programming 'GLOBAL' and 'IMPORT' are interchangeable. In program listings 'IMPORT' is used.

Type:

String operator

Purpose:

To check whether one text string is contained in another.

Syntax:

⟨expression1⟩ IN ⟨expression2⟩

Execution:

A check is made to see whether ⟨expression1⟩ is contained in ⟨expression2⟩. If it is, the logical value returned will be true (= 1). If it is not, the logical value returned will be false (= 0).

Example:

```
10 DIM A$ OF 15
20 DIM B$ OF 15
30 INPUT "WRITE A TEXT:     ": A$
40 INPUT "WRITE ANOTHER TEXT: B$
50 IF B$ IN A$ THEN
60  PRINT "SECOND TEXT IS PART OF FIRST TEXT"
70 ELSE
80  PRINT "SECOND TEXT IS NOT PART OF FIRST TEXT"
90 ENDIF
```

Type:

System variable

Purpose:

To define the number of character positions to be used for the indent of the internal part of structures in program listings.

Syntax:

INDENTION

Execution:

The actual value of the system variable 'INDENTION' controls the number of character positions the internal part of a structure is indented in program listings.

Examples:

```
100 INDENTION:=8
100 INDENTION=(A+3)*B
100 PRINT INDENTION
    INDENTION:=3
```

Comments:

1. On loading COMAL-80 'INDENTION' is assigned the value of 2. This value can be changed by assignment of a new value to 'INDENTION'.

2. Any assigned value must be an integer number between 0 and 10 inclusive. The assigned value will be rounded if necessary.

3. Assignments can be made to this keyword and it can also be used as operand. When used as operand it is of integer type.

4. The 'NEW' command does not change the value of the system variable 'INDENTION'.

Type:

Statement, command

Purpose:

To prepare a formatted diskette (in a drive) for use.

Syntax:

INIT [(device)]

Execution:

The (device) stated is initialized.

Examples:

```
100 INIT "DK0:"
    INIT
    INIT DK1:
```

Comments:

1. Under CP/M, all disk drives will be initialized and the (device) indication is not used. If used, (device) must be the name of a valid disk drive. No disk files may be open when this statement/command is executed.

2. (device) must be stated when 'INIT' is used as a statement, but may be the empty string.

Type:

Machine code function

Purpose:

To read the value at a Z-80 microprocessor input port.

Syntax:

INP(⟨expression⟩)

Execution:

The input port, defined by ⟨expression⟩ is read and the value found there is returned.

Example:

```
10 PRINT INP(17)
```

Comments:

1. ⟨expression⟩ must be between 0 and 255 (inclusive).

2. ⟨expression⟩ will be rounded to integer form if necessary.

Type:

Statement

Purpose:

To read and to assign to variables the values input through the terminal during program execution.

Syntax:

INPUT [(text):] (variable list)

Execution:

When meeting the 'INPUT' statement, program execution pauses after displaying an optional (text). As the user keys in values, they are assigned to the stated variables in (variable list) from left to right. Having inserted the last value the user presses 'RETURN' and program execution continues.

Examples:

```
100 INPUT MONKEY, JOHN#, NAME$
100 INPUT "WRITE 3 DIGITS: ": A, B, C
```

Comments:

1. If the 'INPUT' statement contains a (text), this is displayed exactly as entered. '?' alone is displayed when there is no (text), indicating that the computer expects an input.

2. If (variable list) ends with a comma, the next output appears in the following print-zone. The width of the print-zones is set by using 'ZONE'.

3. If (variable list) ends with a semicolon, the next output appears immediately after the last entry.

4. Several numeric values may be entered as long as they are separated by a character. This character cannot be part of a numerical value such as space or comma.

5. String constants must be entered as a sequence of ASCII characters. It is only possible to insert values following a string constant if the 'RETURN' key is used to terminate each one.
   When a string constant follows an arithmetic constant, COMAL-80 considers the first character (which may not be part of the arithmetic constant), a delimiter and then the string constant with the next character.

6. The type of values keyed in must conform to the types stated in the 'INPUT' statement.

7. (variable list) may contain any variable type, but arrays must be properly indexed and substrings may not be used.

8. Responding to 'INPUT' with the wrong type of value leads to the error message 'ERROR IN NUMBER' and the item must be corrected. No assignment is made until an acceptable input is given.

9. Responding to 'INPUT' with too few items, causes a '?' to be printed on the terminal and the program awaits more input.

10. Responding to 'INPUT' with too many items causes the error message 'TOO MUCH INPUT' and the input must then be corrected.

11. 'INPUT' statements are not allowed in functions.

Type:

Statement

Purpose:

To read data from an ASCII data-file written by the 'PRINT (USING) FILE' statement.

Syntax:

INPUT FILE (file No.) [,(rec. No.)]:(variable list)

Execution:

The values of the variables in (variable list) are read from the file contained in (file No.).

Examples:

```
100 INPUT FILE 3: A$
100 INPUT FILE 0: B#, C
```

Comments:

1. Before meeting the 'INPUT FILE' statement a file must be opened and the connection established between the stated file name and the (file No.) of the 'INPUT FILE' statement. This is done using the 'OPEN FILE' statement or command, followed by 'READ' or 'RANDOM'.

2. The (rec. No.) is used only in 'RANDOM' files and is an arithmetic expression which is rounded to an integer value if necessary.

3. (file No.) is an arithmetic expression.

4. (variable list) may contain all variable types but arrays must be properly indexed and substrings may not be used.

5. The elements of (variable list) are separated by commas.

6. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.

7. Comments 4, 5, 6, and 11 to the 'INPUT' statement apply here as well.

Type:

Arithmetic function

Purpose:

Returns the largest integer equal to or less than the specified expression.

Syntax:

INT(⟨expression⟩)

Execution:

The largest integer less than or equal to ⟨expression⟩ is calculated.

Example:

```
10 INPUT A
20 B:=INT(A)
30 PRINT B
40 PRINT INT(5.72)
50 PRINT INT(-5.72)
```

Comments:

1. ⟨expression⟩ is of real type. The result is an integer of real type.

2. See also the 'ROUND' and 'TRUNC' functions.

Type:

Arithmetic function

Purpose:

To convert an integer, from a string to an integer of integer type.

Syntax:

IVAL(<string expression>)

Execution:

The characters in <string expression>, which must represent a valid integer number, are converted to integer numeric form.

Example:

```
10 DIM A$ OF 4
20 INPUT A$
30 PRINT IVAL(A$)
40 PRINT IVAL("3215")
```

Comments:

1. If the string in <string expression> contains other characters than digits (including a sign), program execution is stopped and an error message is displayed.

2. Also see the 'VAL' function.

Type:

System variable

Purpose:

To specify whether keywords in program listings should appear in upper or lower case letters.

Syntax:

KEYWORDLOWER

Execution:

The current value of the system variable 'KEYWORDLOWER' controls the format of keywords in program listings.

Examples:

```
100 KEYWORDLOWER:=0
100 KEYWORDLOWER:=A
100 KEYWORDLOWER:=TRUE
100 PRINT KEYWORDLOWER
    KEYWORDLOWER:=1
```

Comments:

1. On loading COMAL-80 'KEYWORDLOWER' is assigned the value of 0. This value can be changed by an assignment to 'KEYWORDLOWER' only.

2. The value assigned must be 0 or 1. Assigned values are rounded if necessary.

3. If the value of 'KEYWORDLOWER' is equal to 0, then all keywords will be listed in upper case. Otherwise they will be listed in lower case.

4. Assignments can be made to this keyword and it can be used as operand. When used as operand it is of integer type.

5. The 'NEW' command does not change the value of the system variable 'KEYWORDLOWER'.

Type:

Statement

Purpose:

To name a point in a COMAL-80 program for reference by the 'GOTO' and 'RESTORE' statements.

Syntax:

LABEL <name>

Execution:

The 'LABEL' statement is non-executable and serves only to mark a point in the program.

Example:

```
10 LABEL START
20 INPUT "WRITE A NUMBER: ": NUMBER
30 PRINT NUMBER
40 GOTO START
```

Comments:

1. A 'LABEL' statement used inside a procedure or function can only be referenced inside this local area.

Type:

Arithmetic function.

Purpose:

Returns the length of a string variable.

Syntax:

LEN(<variable>)

Execution:

The number of characters in <variable> is counted.

Example:

```
10 DIM A$(1:10) OF 15
20 INPUT A$(5)
30 B#:=LEN(A$(5))
40 PRINT A$(5)
50 PRINT B#
```

Comments:

1. The current contents of the <variable> are used to determine its length. The dimensioned length only is of importance, since it is the maximum value of the result.

Type:

Statement

Purpose:

To assign the value of an expression to a variable.

Syntax:

[LET] ⟨variable⟩ := ⟨expression⟩

Execution:

⟨expression⟩ is calculated and the result is stored in the space allocated for that ⟨variable⟩.

Example:

```
10 LET A := 5
20 LET B := 3
30 LET SUM := A+B
40 A:+B
50 DIFFERENCE := A-B
60 PRINT SUM
70 PRINT A
80 PRINT DIFFERENCE
```

Comments:

1. The use of the word 'LET' is optional, it may be omitted as shown in line 40 of the example. In program listings 'LET' is omitted.

2. During programming '=' and ':=' are interchangeable. In program listings ':=' is used.

3. ⟨variable⟩ := ⟨variable⟩ + ⟨expression⟩ may be written as ⟨variable⟩ :+ ⟨expression⟩.
   ⟨variable⟩ := ⟨variable⟩ - ⟨expression⟩ may be expressed ⟨variable⟩ :- ⟨expression⟩, though the latter may not be used for string variables.

4. The types used for ⟨expression⟩ and ⟨variable⟩ must be the same. Integer values can be assigned to a real variable.

5. For string variables having ⟨expression⟩ longer than ⟨variable⟩, ⟨expression⟩ will be truncated from the right.

6. For string variables having ⟨expression⟩ shorter than ⟨variable⟩, ⟨variable⟩ takes the actual length only.

7. When assigning to substrings, ⟨expression⟩ and ⟨variable⟩ must be of the same length.

8. Several assignments may be performed on a single line separated by semicolons, and the reserved word 'LET' (which is optional) may only appear in front of the first assignment.

Type:

Command

Purpose:

To list a program in ASCII, in full or in part.

Syntax:

```
LIST [<start>][,<end>][<file name>]
LIST [<start>,][<file name>]
```

Execution:

The specified part of of the program is converted from its internal format to a string of ASCII characters and listed on the specified file (or device).

Examples:

```
LIST
LIST 10
LIST 10,100
LIST ,100
LIST 100,
LIST TEST
LIST 10,100 TEST
LIST ,100 DK1:TEST
LIST LP0:
```

Comments:

1. If <file name> is omitted all listings are displayed on the terminal carrying the device name 'DS0:'.
   If the specified listing contains more lines than the device is able to show in one screen, only the first page is shown and the COMAL-80 interpreter waits for the 'SPACE BAR' to be pressed before displaying the next page, or the 'RETURN' key to display the next line. Pressing the 'ESC' key will end the listing.

2. Omitting both <start line> and <end line> lists the entire program. Omitting only <start line>, causes the listing to start at the first program line. Leaving <end line> out continues the listing to the end of the program. Specifying only <start line>, without the comma, lists only the specified line.

3. The 'LIST' command considers all listings as being a transfer of characters from the memory to a file. Consequently, a listing on a connected printer is obtained by stating 'LP:' for a <file name>, obtionally followed by the unit number of the printer. When no unit number is specified it defaults to LP0:.

4. Listings may not necessarily have the same form as when originally keyed in since automatic indenting takes place in order to clarify the program structure. 'LABEL' statements are not indented making them easy to find. When several keywords have identical meanings, only one of them is used for all listings.

5. If (file name) does not contain an extension it defaults to '.CML'.

6. Programs stored using the 'LIST' command may be read later using the 'ENTER' command.

7. Programs intended for storage for a longer period of time, and programs intended for exchange, should be stored using 'LIST' command as this format is compatible with all COMAL-80 versions and future versions will attempt to follow this.

8. If (file name) is already on the device in question this is reported and the user is offered the option of continuing and having the old file deleted, or of stopping ('RETURN/ESC').

9. The amount of indent can be selected by means of 'INDENTION'.

Type:

Command

Purpose:

To read a binary file from the background storage device.

Syntax:

LOAD <file name>

Execution:

The working memory of the computer is cleared, the operating system is called and the file is read.

Examples:

```
LOAD TEST
LOAD DK1:PROGRAM
```

Comments:

1. Only binary files can be read by the 'LOAD' command, i.e. files stored by the 'SAVE' command. In catalog listings these files can be identified by the extension '.CSB'.
2. The extension '.CSB' is always supplied by the COMAL-80 system and cannot be entered by the user.
3. Any 'EXTENSION' which was present when the program was 'SAVE'd must also be present when the program is 'LOAD'ed again.
4. Before 'LOAD'ing, an implicit 'NEW' command is automatically executed.

Type:

Arithmetic function

Purpose:

Returns the natural logarithm of an arithmetic expression.

Syntax:

LOG(⟨expression⟩)

Execution:

The natural logarithm of ⟨expression⟩ is calculated.

Examples:

```
10 INPUT A
20 PRINT LOG(A)
```

Comments:

1. ⟨expression⟩ may be an arithmetic expression of real or integer type. The result will always be real.

2. If ⟨expression⟩ is less than or equal to 0, program execution is stopped and followed by an error message.

Type:

Statement, command

Purpose:

To terminate logging mode and close the log file.

Syntax:

LOGOFF

Execution:

The logging mode is terminated and the file is closed.

Examples:

```
100 LOGOFF
    LOGOFF
```

Type:

Statement, command

Purpose:

To produce a log of everything displayed on the screen.

Syntax:

LOGON (file name)

Execution:

A file with the given (file name) is opened and everything which COMAL-80 sends to the screen is written to this file.

Examples:

```
100 LOGON "LOGFILE.LOG"
    LOGON LOGFILE
    LOGON LP:
```

Comments:

1. When the log is stored in a disk file it can be displayed on the screen by the program

```
10 DIM A$ OF 160
20 OPEN FILE 0, "LOGFILE.LOG", READ
30 REPEAT
40  INPUT FILE 0: A$
50  PRINT A$
60 UNTIL EOF(0)
70 CLOSE
```

2. If logging is stopped through a 'LOGOFF' statement and then restarted under the same (file name) the new information is appended to the file.

Type:

Statement

Purpose:

To repeat execution of a program section until an internal condition has been fulfilled.

Syntax:

```
LOOP
.
.
.
ENDLOOP
```

Execution:

The program section enclosed by 'LOOP...ENDLOOP' is executed repeatedly until an 'EXIT' statement is encountered. Program execution resumes at the first executable line following the 'ENDLOOP' statement.

Example:

```
10 NUMBER:=0
20 LOOP
30  NUMBER:+1
40  PRINT NUMBER
50  IF NUMBER=8 THEN EXIT
60 ENDLOOP
```

Comments:

1. The execution of the 'LOOP...ENDLOOP' section may be interrupted by a 'GOTO' statement.

2. If 'LOOP...ENDLOOP' statements are nested, execution of an 'EXIT' statement will abandon execution of the innermost 'LOOP...ENDLOOP' statement containing the 'EXIT' statement only.

Type:

Statement

Purpose:

To assign values to all elements in an array.

Syntax:

MAT ⟨variable⟩:=⟨expression⟩

Execution:

Each element in ⟨variable⟩ is assigned the value of ⟨expression⟩.

Example:

```
10 DIM ARRAY(50)
20 MAT ARRAY:=5
```

Comments:

1. ⟨variable⟩ and ⟨expression⟩ must be of the same type. However, an integer expression may be assigned to the elements in a real array.

2. For string variables having ⟨expression⟩ longer than ⟨variable⟩, ⟨expression⟩ will be truncated from the right.

3. For string variables having ⟨expression⟩ shorter than ⟨variable⟩, ⟨variable⟩ takes the current length only.

4. Several assignments may be made on a single line (separated by semicolons), but the keyword 'MAT' may only appear in front of the first assignment.

5. During programming '=' and ':=' are interchangeable. In program listings ':=' is used.

Type:

Arithmetic operator

Purpose:

To return the remainder following an integer division.

Syntax:

⟨expression1⟩ MOD ⟨expression2⟩

Execution:

⟨expression1⟩ is integer divided by ⟨expression2⟩. The remainder is ⟨expression1⟩ minus the result, multiplied by ⟨expression2⟩.

Example:

```
10 INPUT A
20 B:=A MOD 7
30 PRINT B
```

Comments:

1. The result N is defined by the lowest non-negative value which the expression:

   ⟨expression1⟩ - N * ⟨expression2⟩

   can assume for integer N.

2. The type of the result depends upon the type of ⟨expression1⟩ and ⟨expression2⟩ in the following way:

⟨expression1⟩	MOD	⟨expression2⟩	result
real		real	real
real		int	real
int		real	real
int		int	int

3. Also see the 'DIV' operator.

Type:

Command

Purpose:

To clear the computer's memory and prepare the COMAL-80 system for a new program.

Syntax:

NEW

Execution:

The stored program (if any) and any variables left over from a previous program execution are deleted and the space is recovered for use by a new program.
In addition, the equivalent to the following program is executed:

```
10 CLOSE
20 SELECT OUTPUT "DS:"
30 TRAP ERR+
40 TRAP ESC+
```

and the system functions 'ERR()' and 'ESC()' will subsequently return 0.

Example:

```
NEW
```

Comments:

1. The 'NEW' command should always be used before keying in a new program.

2. System variables ('KEYWORDLOWER', 'IDENTIFIERLOWER', 'INDENTION', 'PAGEWIDTH', 'PAGELENGTH' and 'ZONE') are not affected by this command.

3. Also see note 2 to the 'ENTER' command.

Type:

Logic operator

Purpose:

To perform the logical 'NOT' operation.

Syntax:

NOT ⟨expression⟩

Execution:

The logical value of ⟨expression⟩ is logically negated.

Example:

```
100 IF NOT ERR() THEN EXEC READ_OK
```

Comments:

1. The operator has the following truth table

⟨expression⟩	result
true	false
false	true

Type:

Statement

Purpose:

To transfer execution to a program line number resulting from the evaluation of an expression.

Syntax:

ON ⟨expression⟩ GOTO ⟨list of line numbers⟩
ON ⟨expression⟩ GOSUB ⟨list of line numbers⟩

Execution:

⟨expression⟩ is evaluated and rounded to integer if necessary. The corresponding line number is chosen from ⟨list of line numbers⟩. ⟨expression⟩=1 corresponds to the first line number from the left; ⟨expression⟩=2 corresponds to the second line number from the left and so on.

Example:

```
10 INPUT "WRITE A NUMBER BETWEEN 1 AND 3 INCL: ": NUMBER
20 ON NUMBER GOTO 40,60,80
30 GOTO 10
40 PRINT "YOU WROTE 1"
50 GOTO FINISH
60 PRINT "YOU WROTE 2"
70 GOTO FINISH
80 PRINT "YOU WROTE 3"
90 LABEL FINISH
```

Comments:

1. Unlike the 'GOTO' statement, names may not be used in the 'ON...GOTO' statement.

2. If the rounded value of ⟨expression⟩ does not fulfil the test:
   1 ⟨= ⟨expression⟩ ⟨= items in ⟨list of line numbers⟩
   the statement is skipped and the program is resumed from the next executable statement.

3. For 'ON...GOSUB' statements each line number in ⟨list of line numbers⟩ must be the first statement in a subroutine ended by a 'RETURN' statement.
   On meeting this, the program execution resumes at the first executable line after the 'GOSUB' statement.

4. See also the 'GOSUB' statement.

Type:

Statement, command

Purpose:

To open a data file on the background storage device.

Syntax:

OPEN FILE ⟨file No.⟩,⟨file name⟩,⟨type⟩[,⟨record size⟩]

Execution:

First, a file with name ⟨file name⟩ is searched for on the background storage device.
If it is found and 'WRITE' ⟨type⟩ was stated, or if it is not found and 'READ' ⟨type⟩ was stated, program execution is stopped and an error message is displayed.
In the case of 'APPEND' or 'RANDOM' ⟨type⟩ being stated, the file is created if not found; otherwise the existing file is used.
Then ⟨file name⟩ and ⟨file No.⟩ are coupled so that all references to ⟨file name⟩ are done by ⟨file No.⟩ until the file is closed with a 'CLOSE' statement or command.

Examples:

```
100 OPEN FILE 2,"TEST",WRITE
100 OPEN FILE 0,"DK1:DATA.RAN",RANDOM,40
```

Comments:

1. ⟨file No.⟩ is an arithmetic expression which must be one of the integer values 0 to 9 inclusive.

2. ⟨file name⟩ must be a string expression. Please note that not all operating systems allow all possible characters in file names. For example, CP/M allows only 8 characters, and only 8 characters are transferred to the disk.

3. ⟨type⟩ specifies how the file is used. The following options are available:

READ	Reads sequentially from the file.
WRITE	Writes sequentially to the file.
RANDOM	Reads from and writes to the file.
APPEND	Appends new information to an existing file created using 'WRITE' ⟨type⟩.

4. ⟨record size⟩ is used only for 'RANDOM' files and expresses the total number of bytes to be written to each record. The necessary size is calculated as follows (assuming that 'READ' and 'WRITE' statements are used when the file is read from and written to):
   - Integers take 2 bytes
   - Real figures take 4 bytes at 7-digit precision, and 8 bytes at 13-digit precision.
   - Strings take 2 bytes plus one byte per character of the string.

5. Up to 8 disk files may be open at one time. This leaves room for another 2 non-disk files to be open at the same time. If disk files are used in connection with 'LOAD', 'SELECT OUTPUT', 'LIST', 'SAVE', 'CAT', or 'ENTER', fewer than 8 disk files may be opened by 'OPEN'.
A file may be open on several file numbers at the same time provided that the same (type) is used.

6. A 'RANDOM' file must always be re-opened using the same (record size) with which it was originally opened.
(record size) can be recovered using the program:

```
10 OPEN FILE 0,"(file name).RAN",READ
20 READ FILE 0: RECORD_SIZE#
30 PRINT RECORD_SIZE#
40 CLOSE
```

Type:

Logical operator

Purpose:

Performs the logical 'OR' operation between two expressions

Syntax:

⟨expression1⟩ OR ⟨expression2⟩

Execution:

⟨expression1⟩ and ⟨expression2⟩ are evaluated and if equal to zero considered false, otherwise true. ⟨expression1⟩ is then ORed with ⟨expression2⟩.

Example:

```
100 IF END_DATA1 OR END_DATA2 THEN EXEC END_DATA
```

Comments:

1. The operator has the following truth table:

⟨expression1⟩	⟨expression2⟩	result
true	true	true
true	false	true
false	true	true
false	false	false

Type:

Arithmetic function

Purpose:

To convert the first character in a string into its ASCII value.

Syntax:

ORD((string expression))

Execution:

Returns the ASCII value of the first character in (string expression).

Example:

```
10 DIM A$ OF 1
20 INPUT A$
30 PRINT ORD(A$)
```

Comments:

1. The result is an integer between 0 and 255.

Type:

Machine language function

Purpose:

To send a byte to a Z80 output port.

Syntax:

OUT⟨expression1⟩,⟨expression2⟩

Execution:

The values of ⟨expression1⟩ and ⟨expression2⟩ are evaluated and rounded if necessary. The value of ⟨expression2⟩ is sent to the machine output port corresponding to ⟨expression1⟩.

Example:

```
10 INPUT A
20 OUT 15,A
```

Comments:

1. The value of ⟨expression1⟩ and ⟨expression2⟩ must be a real or integer number between 0 and 255.

2. Also see 'INP'.

Type:

Statement, command

Purpose:

To advance the paper on a printer to the top of the next page.

Syntax:

PAGE

Execution:

If 'PAGELENGTH' = 0, a form feed character is transmitted to the line printer. Otherwise, the line feed character (0AH) is transmitted until the top of the next page is reached.

Examples:

```
100 PAGE
    PAGE
```

Comments:

1. Form feed is controlled by a counter within COMAL-80 when 'PAGELENGTH' > 0. In this case, it is important that the paper is inserted correctly in the printer and that it is not fed manually.

2. The length of a page can be changed by the 'PAGELENGTH' statement or command.

Type:

System variable

Purpose:

To specify the number of lines per page on an attacned printer.

Syntax:

PAGELENGTH

Execution:

An internal counter in COMAL-80 keeps track of the number of lines printed on the current page on the printer. This number is used when a 'PAGE' statement or command is executed and a form feed character is sent to the printer which together with the value of 'PAGELENGTH' determines tne number of line feed characters to be substituted for tne form feed character. Thus, 'PAGELENGTH' determines the number of lines on a page.

Examples:

```
100 PAGELENGTH:=72
100 PRINT PAGELENGTH
100 PAGELENGTH:=MAX_LINES
    PAGELENGTH=88
```

Comments:

1. On loading COMAL-80 'PAGELENGTH' is assigned the value of 72. This value can be changed by an assignment to 'PAGELENGTH'.

2. An assigned value must lie between 0 and 254 (inclusive)

3. This keyword can be used as operand or may be assigned to. When used as operand it is of integer type.

4. The current value of 'PAGELENGTH' is valid for both possible printers.

5. The value 0 stops the internal counter and disables the translation of form feed into line feed characters. Thus form feed characters can be sent to the printer as such.

6. The 'NEW' command does not change the value of the system variable 'PAGELENGTH'.

Type:

System variable

Purpose:

To specify the number of characters per line on an attached printer.

Syntax:

PAGEWIDTH

Execution:

An internal counter in COMAL-80 keeps track of the current print position and issues a carriage return and line feed command when the maximum allowed number of characters has been printed.

Examples:

```
100 PAGEWIDTH:=40
100 PRINT PAGEWIDTH
100 PAGEWIDTH:=MAX_CHARACTERS
    PAGEWIDTH:=80
```

Comments:

1. On loading COMAL-80 'PAGEWIDTH' is assigned a value of 80. This value can be changed by an assignment to 'PAGEWIDTH'.

2. An assigned value must lie between 0 and 254 (inclusive)

3. This keyword can be used as operand or may be assigned to. When used as operand it is of integer type.

4. The current value of 'PAGEWIDTH' is valid for both possible printers.

5. The value 0 inhibits the automatic issuing of carriage return and line feed.

6. The 'NEW' command does not change the value of the system variable 'PAGEWIDTH'.

Type:

Machine language function

Purpose:

To determine the value of a memory location determined by an arithmetic expression.

Syntax:

PEEK(<expression>)

Execution:

The value of <expression> is evaluated and rounded if necessary. The value of the corresponding memory address is returned.

Example:

```
 10 DIM B$ OF 1
 20 TRAP ESC-
 30 EXEC GET_CHR_ESC(B$)
 40 PRINT B$
 50 PROC GET_CHR_ESC(REF A$)
 60  // GET KEYBOARD INPUT WITHOUT ECHO TO SCREEN
 70  // THE 'ESC' KEY IS TREATED LIKE ANY OTHER
 80  // CHARACTER.
 90  // THE 'TRAP ESC-' STATEMENT MUST BE EXECUTED BEFORE
100  // THIS PROCEDURE IS CALLED.
110  POKE 256, 255
120  REPEAT
130   IF ESC() THEN POKE 256, 27
140  UNTIL PEEK(256)<>255
150  A$:=CHR$(PEEK(256))
160 ENDPROC GET_CHR_ESC
```

Comments:

1. The value of <expression> must be a real or integer number between 0 and 65535. The result will be of integer type between 0 and 255.

2. Also see 'POKE'.

Type:

Machine language function

Purpose:

To set the contents of a memory location to a value determined by an arithmetic expression.

Syntax:

POKE⟨expression1⟩,⟨expression2⟩

Execution:

The values of ⟨expression1⟩ and ⟨expression2⟩ are evaluated and rounded if necessary. The memory address corresponding to ⟨expression1⟩ is loaded with the value of ⟨expression2⟩.

Example:

```
 10 DIM B$ OF 1
 20 EXEC GET_CHARACTER(B$)
 30 PRINT B$
 40 PROC GET_CHARACTER(REF A$)
 50  // GET KEYBOARD INPUT WITHOUT ECHO ON THE SCREEN
 60  // THE 'ESC' KEY WORKS IN THE NORMAL WAY
 70  POKE 256, 255
 80  REPEAT
 90  UNTIL PEEK(256)<>255
100  A$:=CHR$(PEEK(256))
110 ENDPROC GET_CHARACTER
```

Comments:

1. The value of ⟨expression1⟩ must be a real or integer number between 0 and 65535. The value of ⟨expression2⟩ must lie between 0 and 255.

2. Also see 'PEEK'.

Type:

Arithmetic function

Purpose:

To determine whether one string is contained within another and, if so, whereabouts.

Syntax:

POS(⟨string expression1⟩,⟨string expression2⟩)

Execution:

A character by character test is made to see if ⟨string expression1⟩ is contained in ⟨string expression2⟩. If it is, the result of the function is an integer returning the character position of ⟨string expression2⟩ at which ⟨string expression1⟩ starts.

Example:

```
10 DIM A$ OF 25
20 DIM B$ OF 25
30 INPUT "FIRST STRING:  ":A$
40 INPUT "SECOND STRING: ":B$
50 C#:=POS(A$,B$)
60 PRINT C#
```

Comments:

1. If ⟨string expression1⟩ is a null string, the function returns the value 1.

2. If ⟨string expression1⟩ is not contained in ⟨string expression2⟩, the function returns the result 0.

3. The result of the function is always an integer.

Type:

Statement, command

Purpose:

To display data on an output device.

Syntax:

PRINT [⟨list of expressions⟩]

Execution:

The ⟨list of expressions⟩ consists of variables, constants, and literals, the values of which are output to the assigned output device.

Examples:

```
100 PRINT "THE RESULT IS: "; A
100 PRINT TAB(15); A, B
```

Comments:

1. The single elements of ⟨list of expressions⟩ must be separated by commas or semicolons. If two elements are separated by a semicolon, the second element is printed immediately after the first one, while a space is inserted after an arithmetic expression. Separating two elements with a comma causes the second element to be printed at the start of the next print-zone.
   The width of the print-zones may be changed using 'ZONE:= ⟨arithmetic expression⟩' executed as a statement or a command for which ⟨arithmetic expression⟩ is rounded to an integer greater than or equal to 0 and less than or equal to 160.
   The rules for semicolon and comma are also valid after the last element in ⟨list of expressions⟩, as the effect is carried onto the first element of the next 'PRINT' statement.
   When ⟨list of expressions⟩ ends without a comma or semicolon, the execution of the statement ends with a line feed.
   This also happens if ⟨list of expressions⟩ is omitted.

2. If the remaining space on the actual line is too short to contain the next print element, it is printed from the start of the following line.

3. Execution of a 'SELECT OUTPUT' statement switches between output devices.

4. ⟨expression⟩ is arithmetic and represents the number of character positions from the left margin, the function 'TAB (⟨expression⟩)' tabulates to the required character position.
   For more details see 'TAB'.

5. During programming 'PRINT' may be replaced by ';'. In program listings 'PRINT' is used.

Type:

Statement

Purpose:

To write data in ASCII format to a data file.

Syntax:

PRINT FILE ⟨file No.⟩[,⟨rec. No.⟩]:⟨list of expressions⟩

Execution:

The values of the expressions in ⟨list of expressions⟩ are written to the file indicated by ⟨file No.⟩.

Examples:

```
100 PRINT FILE 0,RECNO: A$, B, C+D

100 DIM A$ OF 5
110 A$:="##.##"
120 PRINT FILE 3: USING "##.##": A, B, C^2
130 PRINT FILE 4: USING A$: D
```

Comments:

1. Before meeting the 'PRINT FILE (USING)' statement, a file must be opened and connection between the ⟨file name⟩ and the ⟨file No.⟩ used in the 'PRINT FILE (USING)' statement must be established with an 'OPEN FILE' statement or command, and a type: 'APPEND', 'WRITE' or 'RANDOM'.

2. ⟨rec. No.⟩ is only needed for 'RANDOM' files and is an arithmetic expression which will be rounded to integer if necessary and which designates the number of the logical record of the file to be used.

3. ⟨file No.⟩ is an arithmetic expression.

4. The elements in ⟨list of expressions⟩ should be separated by commas or semicolons, similar to the syntax of 'PRINT' and 'PRINT USING'.

5. 'PRINT FILE' and 'PRINT FILE USING' perform similar functions to 'PRINT' and 'PRINT USING', the only difference being the destination of the output.
   The syntax for 'PRINT FILE USING' is obtained by substituting ⟨list of expressions⟩ in the above syntax with:
   USING ⟨string expression⟩:⟨list of expressions⟩

6. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.

7. During programming 'PRINT' may be replaced by ';'. In program listings 'PRINT' is used.

Type:

Statement

Purpose:

To print text strings and/or numbers in a specified format.

Syntax:

PRINT USING ⟨string expression⟩:⟨list of expressions⟩

Execution:

The text string specified in ⟨string expression⟩ is transferred character by character onto the output device. String expressions and/or arithmetic expressions from ⟨list of expressions⟩ are used to replace the '#' characters.

Examples:

```
100 PRINT USING "THE RESULT IS ###.##": A

 10 DIM A$ OF 6
 20 A$:="##.###"
 30 PRINT USING A$: B
```

Comments:

1. The individual characters in ⟨string expression⟩ have the following significance:
   '#' character position and sign.
   '.' decimal point if surrounded by '#'.
   '+' preceding plus, when followed by '#'.
   '-' preceding minus, when followed by '#'.
   All other characters are transferred unchanged.

2. A format starting with '+' will assign a space for sign which will be printed for both negative and positive values.

3. A format starting with '-' will assign space for signs but it will be printed for negative values only.

4. For text strings a preceding '+' or '-' will be treated as '#'.

5. If an arithmetic value contains too many digits to be printed in the specified format, the position is filled with '*'. If an arithmetic value contains more decimals than specified in the format, rounding takes place automatically.

6. Text strings always start at the extreme left within the format. If a string is too long, the necessary number of characters is deleted from the right. When a text string is too short, the rest of the format is filled with spaces.

7. When there are no more expressions in (list of expressions), execution of the 'PRINT USING' statement is terminated. If (list of expressions) contains more expressions than stated in (string expression), the formats within are again used from the left.

8. If the 'PRINT USING' statement ends with a semicolon, the next printout will start immediately after the output produced by the 'PRINT USING' statement. If it ends with a comma the next printout will start at the beginning of the next print zone. Otherwise the execution of the 'PRINT USING' statement will cause a change to a new line.

9. The 'PRINT USING' statement may be used for writing in a data file following exactly the same rules as described for the 'PRINT FILE' statement.

10. During programming 'PRINT' may be replaced by ';'. In program listings 'PRINT' is used.

Type:

Statement

Purpose:

To define and name a sub-program (a procedure)

Syntax:

```
PROC (name)[(formal parameter list)] [CLOSED]
.
.
ENDPROC (name)
```

Execution:

On encountering a 'PROC' statement, the program section is skipped up to and including the corresponding 'ENDPROC' statement. It will be executed only when the procedure is called by name using an 'EXEC' statement.

Examples:

```
10 PROC ERROR(N#) CLOSED
20  IMPORT CC#, ERR_, ERRORS#
30  PRINT "****";SPC(CC#-9);"^";N#
40  ERR_:=INCLUDE(ERR_,N#+1); ERRORS#:+1
50 ENDPROC ERROR

PROCEDURE HEADINGS ONLY:
10 PROC XYZ(A,B,REF C$) CLOSED
10 PROC ZYX(REF A#(,,), REF C(), D$)
10 PROC YZX(REF D$(,,), REF E#, REF C) CLOSED
```

Comments:

1. The 'PROC' statement may not be used within:
   - Conditional statements
   - Repeating statements
   - Other procedure or function declarations

2. (name) must be a legal variable name.

3. A procedure may call other functions, and even itself (recursion). A closed procedure can only call a closed function or procedure.

4. (formal parameter list) contains the names of the formal parameters which will receive values from the current parameters in the procedure call when called.

5. The changes to a parameter within a procedure remain local unless 'REF' is used to indicate that the changes should affect the actual parameter.

6. 'REF' may be stated for simple arithmetic or string variables, and must be stated for all array variables.

7. A procedure type may be either real, integer or string.

8. Array variables must be followed by a dimension definition consisting of commas in parentheses corresponding to the number of dimensions -1. I.e. for 3-dimensional arrays, the parenthesis contains two commas, while a vector would be followed by an empty parenthesis.

9. If the procedure is declared 'CLOSED' variable names remain local and may be used for other purposes outside the procedure. This may be declared invalid for one or more variables using the 'IMPORT' statement.

10. If the program section between 'PROC' and 'ENDPROC' contains statements of multiple lines these must all be contained in the program section.

11. As well as using an 'ENDPROC (name)' statement to return from the procedure, it is also possible to use the 'RETURN' statement.

12. The sections 'PROCEDURES' and 'PARAMETER SUBSTITUTION' in chapter 1 give a more detailed description of these keywords.

Type:

Statement, command

Purpose:

To stop the COMAL-80 interpreter and return to the environment from which it was called.

Syntax:

QUIT

Execution:

Under CP/M a warm boot is performed.

Examples:

```
100 QUIT
    QUIT
```

Type:

Statement, command

Purpose:

To set a random startpoint for the 'RND' functon.

Syntax:

RANDOM
RANDOMIZE

Execution:

The Z-80 CPU has a built-in counter which is read and the value found is used as the seed for the algorithm which calculates the random value.

Examples:

```
100 RANDOM
    RANDOM
```

Comments:

1. The counter works constantly when the the CPU is active. Its clock frequency is around 500 KHz at a CPU clock frequency of 2.5 MHz.

2. If 'RANDOM' is not found in a program calling the 'RND' function, any execution of the program will give the same sequence of random numbers.

3. 'RANDOM' and 'RANDOMIZE' are interchangeable. In program listings 'RANDOM' is used.

Type:

Statement

Purpose:

To assign values to variables from a data list.

Syntax:

READ (variable list)

Execution:

The single elements of (variable list) are assigned values from the data list. This is done in sequence from left to right.

Examples:

```
10 DIM FIRST_NAME$ OF 10
20 DIM FAMILY_NAME$ OF 10
30 DATA "JOHN", "DOE", 10
40 READ FIRST_NAME$, FAMILY_NAME$
50 PRINT FIRST_NAME$+" "+FAMILY_NAME$
60 READ AGE
70 PRINT AGE; "YEAR"
```

Comments:

1. If the type of value does not correspond to that of the stated variable, or if the data list is empty, program execution is stopped with an error message.

2. Assigning values to a string variable follows the same rules as given for 'LET' statements.

3. See the 'DATA' statement.

Type:

Statement

Purpose:

To read data from a binary data file written using the 'WRITE FILE' statement.

Syntax:

READ FILE (file No.) [,(rec No.)]:(variable list)

Execution:

The values of the variables in (variable list) are read from the file connected to (file No.).

Examples:

```
100 READ FILE 5,REC_NO: A
100 READ FILE 3: A, B, C
```

Comments:

1. Before encountering the 'READ FILE' statement, a file must be opened and the connection must be established between the file name and the (file No.) of the 'READ FILE' statement. This is done with the 'OPEN FILE' statement or command and type 'READ' or 'RANDOM'.

2. The (rec No.) is only used in 'RANDOM' files and is an arithmetic expression which will be rounded to integer if necessary. It indicates the number of the logical record to be utilized.

3. (file No.) is an arithmetic expression.

4. (variable list) may contain any variable type. Arrays are read in total if no indices are specified.

5. The elements of (variable list) are separated by commas.

6. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.

Type:

Statement

Purpose:

To transfer variables from the current program to a program called using the 'CHAIN' statement.

Syntax:

RECEIVE (list of variables)

Execution:

When the 'CHAIN' statement, which loads the program containing the 'RECEIVE' statement, is executed, the current values of the variables listed in the 'CHAIN' statement are saved.
The 'RECEIVE' statement is used to enter these values to the new program. After its execution, the variable names in the (list of variables) have been dimensioned appropriatepriately if necessary and have been assigned the values which were saved.

Examples:

```
100 RECEIVE A,B,C
100 RECEIVE A$,B#,C
```

Comments:

1. The type of variables specified in (list of variables) in the 'RECEIVE' and 'CHAIN' statements must correspond.

2. Variables representing arrays and strings carry their dimension from the old to the new program part and must not be re-dimensioned.

Type:

Statement, command

Purpose:

To check that all disk files are closed.

Syntax:

RELEASE [(device)]

Execution:

All disk files are checked to see that they are closed.

Examples:

```
100 RELEASE ""
100 RELEASE "DK1:"
100 RELEASE "DK"+DISK$+":"
    RELEASE
    RELEASE DK1:
```

Comments:

1. Under CP/M, the (device) indication is not used but, if it is given, it must be the name of a disk drive.

2. If a disk file is open, execution is terminated and an error message is displayed.

3. (device) must be given when 'RELEASE' is used as a statement but may be the empty string.

Type:

Statement

Purpose:

To allow for insertion of explanatory text in a COMAL-80 program.

Syntax:

```
//
REM
!
```

Execution:

The 'REM' statement is ignored during program execution.

Examples:

```
10 //PROGRAM TO CALCULATE
20 REM POLYNOMIAL
30 ! 30/10/1980
40 OPEN FILE 4,"TEST",READ //OPEN DATA FILE
```

Comments:

1. During programming 'REM', '//', and '!' are interchangeable. In program listings '//' is used.

2. All statements may be followed by a comment.

Type:

Statement, command

Purpose:

To change the name of a file on the background storage device.

Syntax:

RENAME <old file name>,<new file name>

Execution:

The operating system of the computer is called and the file named <old file name> is renamed to <new file name>.

Examples:

```
220 RENAME "DK1:FIL.CML","DK1:FIL.BAK"
    RENAME DK1:FIL.CML,DK1:FIL.BAK
    RENAME FIL.CML,FIL.BAK
```

Comments:

1. <old file name> must exist on the stated device.

2. If no device is stated, the statement/command is carried out on the current default device.

3. If the <new file name> is already in use, this is reported and the statement/command is terminated.

4. If a device description is contained in one of the names, the same device indication must be part of the other name.

Type:

Command

Purpose:

To renumber program lines and to re-arrange program structures.

Syntax:

RENUM [[<start line>:<end line>,]<start>[,<step>]]

Execution:

If only a part of a program is to be renumbered, a check is made to see whether there is sufficient room to renumber using the intervals specified. If not, execution is stopped followed by an error message.
If there is enough room, the new line numbers are calculated and stored. The program is checked and all references ('GOTO', 'GOSUB', etc.) are updated.
Finally, the old line numbers are deleted.

Examples:

```
RENUM
RENUM 15
RENUM 15,3
RENUM 20:90,310,1
```

Comments:

1. If <step> is not given, default 10 is used.

2. If <start> is not given, default 10 is used.

3. <start line> and <end line> are used when only a section of a program is renumbered. They specify the first and last line numbers to renumber. In this case <start> specifies the first new line number and <step> the new step between line numbers. In this way a program section can be moved to any place in a program if there are enough free line numbers. No overwriting and no mixing is possible.

4. If <start line>:<endline>, is not given the whole program is renumbered.

Type:

Statement

Purpose:

To repeat the execution of a program section until the condition contained in the 'UNTIL' statement is fulfilled.

Syntax:

```
REPEAT
.
.
.
UNTIL (logical expression)
```

Execution:

On meeting the 'UNTIL' statement the value of the (logical expression) is calculated. If it is true, execution resumes from the first executable statement following the 'UNTIL' statement. If (logical expression) is false the program continues from the first executable statement following the 'REPEAT' statement.

Example:

```
 10 DIM A$ OF 1
 20 DIM B$ OF 25
 30 PRINT "THE PROGRAM IS STOPPED BY"
 40 PRINT "PRESSING THE 'ESC' KEY"
 50 TRAP ESC-
 60 REPEAT
 70  INPUT "WRITE A LETTER: ": A$,
 80  B$:=B$+A$
 90 UNTIL ESC()
100 PRINT "YOU WROTE: "; B$
```

Comments:

1. A program section surrounded by 'REPEAT...UNTIL' is always executed at least once.

Type:

Statement

Purpose:

To move the data list pointer to allow it to be partially or wholly re-read.

Syntax:

```
RESTORE (line number)
RESTORE (name)
RESTORE
```

Execution:

The pointer of the data list is set to the first data item in the stated line, or to the first data item declared if no line is specified.

Example:

```
 10 LABEL AGAIN
 20 RESTORE DATA2
 30 READ X
 40 PRINT X
 50 DATA 47
 60 RESTORE 50
 70 READ X
 80 PRINT X
 90 GOTO AGAIN
100 LABEL DATA2
110 DATA -47
```

Comments:

1. If the 'RESTORE' statement contains a line number, the corresponding line must contain a 'DATA' statement.

2. If the 'RESTORE' statement contains a name, the statement immediately following the 'LABEL' statement defining that label must contain a 'DATA' statement.

3. If the 'RESTORE' statement contains neither a line number nor a name, the pointer is set to the first item of the first 'DATA' statement.

Type:

Statement

Purpose:

To terminate a subroutine or a procedure, or to terminate a user defined function and return the function value.

Syntax:

```
RETURN                    (for procedures and subroutines)
RETURN (expression)       (for functions)
```

Execution:

Execution of procedures and subroutines is terminated and resumes from the line following the calling line. For functions, execution is terminated and the function value is inserted in the expression which caused execution of the function.

Examples:

```
10 FUNC X_Y_POWER(X,Y)
20  RETURN X^3/Y^2
30 ENDFUNC X_Y_POWER
40 I:=2
50 J:=3
60 OLE:=X_Y_POWER(I,J)
70 PRINT OLE
```

```
10 PRINT "MAIN PROGRAM"
20 GOSUB 50
30 STOP
50 PRINT "SUBROUTINE"
60 RETURN
```

```
10 EXEC OPEN_FILE
20 PROC OPEN_FILE
30  IF A$="DEFAULT" THEN RETURN
40  OPEN FILE 3,"DK1:"+A$,READ
50 ENDPROC OPEN_FILE
```

Comments:

1. In user defined functions, the function value can only be returned using the 'RETURN' statement. If this is not included, the function value will be undefined and an error message will be displayed.

2. (expression) in the 'RETURN' statement must of be the same type as the function name. The only exception is that an integer expression will be accepted in a function of real type.

3. Within a procedure a 'RETURN' statement without (expression) cannot be used to return from a subroutine. In the main program a 'RETURN' statement can only be used to return from a subroutine.

Type:

Arithmetic function.

Purpose:

To return a pseudo-random number.

Syntax:

RND [()]
RND ⟨expression1⟩,⟨expression2⟩

Execution:

A random number is generated based on the seed (which can be changed with the 'RANDOM' statement/command) or on the most recently generated random number.

Example:

```
100 A:=RND()
100 B:=RND(-5,17)
```

Comments:

1. Any execution of a program will give the same sequence of random figures unless a 'RANDOM' statement has first been executed.

2. Omitting the two limits ⟨expression1⟩ and ⟨expression2⟩ creates a random real number in the range 0 to 1.

3. If ⟨expression1⟩ and/or ⟨expression2⟩ is not an integer, then rounding takes place.

4. If limits are stated, the result will always be an integer between ⟨expression1⟩ and ⟨expression2⟩ inclusive.

5. During programming the parenthesis after 'RND' may be omitted if empty. Thus, instead of 'RND()', 'RND' may be used. In program listings 'RND()' will be used.

Type:

Arithmetic function

Purpose:

To convert a real expression to an integer type.

Syntax:

ROUND((expression))

Execution:

The arithmetic (expression) is rounded and the result is converted to integer type.

Example:

```
10 INPUT A
20 B#:=ROUND(A)
30 C:=ROUND(A)
40 PRINT B#, C
50 PRINT ROUND(5.72)
60 PRINT ROUND(-5.72)
```

Comments:

1. Rounding is carried out to the nearest integer. If the number lies evenly between two integers, the one with the highest absolute value is chosen.

2. (expression) is of real type. The result is an integer type. Note that an integer can be assigned to a real variable.

3. See the 'INT' and 'TRUNC' functions.

Type:

Command

Purpose:

To start execution of a program.

Syntax:

RUN [<line number>]

Execution:

COMAL-80 is brought to a defined start position which, among other things, closes all files left open from any previous execution, performs a 'SELECT OUTPUT "DS:"' and initializes the variable area.
After this a special prepass module checks to see whether the program contains structures (FOR...NEXT, LOOP...ENDLOOP etc.) and references (EXEC, LABEL, etc.) and the internal representation of these statements is extended to increase the working speed.
Finally, program execution is started at the given line number.

Examples:

```
RUN
RUN 230
```

Comments:

1. Omitting <line number> starts the program at the lowest line number.

Type:

Command

Purpose:

To store programs on the background storage device in the internal (binary) format.

Syntax:

SAVE <file name>

Execution:

The operating system of the computer is called and information on <file name> and the area of memory to be transferred is passed to it for the 'SAVE' operation.

Examples:

```
SAVE TEST
SAVE DK1:TEST
```

Comments:

1. If a program is to be called by the 'CHAIN' statement it must have been stored using the 'SAVE' command.

2. Programs stored using the 'SAVE' command may be re-read by the 'LOAD' command.

3. The internal format may be different on different versions of COMAL-80. Consequently, a program cannot always be stored using the 'SAVE' command in one version and read using the 'LOAD' command in an other version. Programs to be exchanged or stored for longer periods of time should be stored using the 'LIST' command.

4. If <file name> already exists on the current device, this is reported and the user may continue, thus deleting the old file, or stop ('RETURN/ESC').

5. The extension '.CSB' is always supplied by the COMAL-80 system and not by the user.

6. Information on the 'EXTENSIONS' loaded at the time of execution of 'SAVE' is also stored in the file. This information is checked when 'LOAD' or 'CHAIN' is used and any discrepancy from the 'EXTENSIONS' loaded at that time is an error.

Type:

Statement, command

Purpose:

To specify a new default device/file for printout by the 'PRINT' and 'PRINT USING' statements.

Syntax:

SELECT [OUTPUT] <string expression>

Execution:

Internal pointers in the COMAL-80 system are switched to select the specified printout device/file.

Examples:

```
220 SELECT OUTPUT "LPO:"
220 SELECT OUTPUT "DK1:TEXT"
220 SELECT OUTPUT "TEXT"
220 SELECT OUTPUT "DS:"
    SELECT OUTPUT "LP:"
```

Comments:

1. Whenever the program execution is started using the 'RUN' command, the console is chosen as default output file.
   During program execution a new default file may be chosen by specifying the name of the peripheral or a file using a <string expression>.
   When program execution is terminated, either by use of the 'ESC' key, or because it is finished, the terminal again defaults as the output file.

Type:

Arithmetic function

Purpose:

Returns the sign of an arithmetic expression.

Syntax:

SGN((expression))

Execution:

Arithmetic (expression) is calculated and if the result is greater than 0 the function returns the value 1. If the result equals 0, 0 is returned, and if the result is less than 0, -1 is returned.

Examples:

```
10 INPUT "WRITE A NUMBER: ": A
20 ON SGN(A)+2 GOTO 30,50,70
30 PRINT "A<0"
40 STOP
50 PRINT "A=0"
60 STOP
70 PRINT "A>0"
80 STOP
```

Type:

Trigonometric function

Purpose:

Returns the sine of an expression.

Syntax:

SIN(⟨expression⟩)

Execution:

The sine of ⟨expression⟩, in radians, is calculated.

Example:

```
10 INPUT A
20 PRINT SIN(A)
```

Comments:

1. ⟨expression⟩ is an arithmetic expression of real or integer type. The result will always be real.

Type:

Command

Purpose:

To display the size of the used memory area.

Syntax:

SIZE

Execution:

The amount of memory used for storage of the user's program with 'EXTENSIONS' is displayed on the terminal, together with the amount remaining and the amount used by variables.

Example:

SIZE

Comments:

1. The figures displayed indicate the number of bytes used or remaining.

2. The figure shown as space used for variables refers only to variables dimensioned or used during the last program execution.

3. The size of COMAL-80 itself is not displayed.

Type:

String function

Purpose:

To create a string consisting of spaces, the number of these being defined by an arithmetic expression.

Syntax:

SPC$(<expression>)

Execution:

The arithmetic <expression> is calculated (and rounded if necessary) then a string containing that number of spaces is created.

Example:

```
10 INPUT A
20 PRINT SPC$(3*5),A
```

Comments:

1. <expression> must be equal to, or greater than, 0.

Type:

Arithmetic function

Purpose:

To calculate the square root of an arithmetic expression.

Syntax:

SQR(⟨expression⟩)

Execution:

The square root of an ⟨expression⟩ equal to or greater than 0 is calculated.

Example:

```
10 INPUT A
20 PRINT SQR(A)
```

Comments:

1. ⟨expression⟩ is arithmetic and may be real or integer. The result will always be real.

2. If ⟨expression⟩ is less than 0, execution is stopped with an error message. If these have been inhibited using the 'TRAP ERR-' statement, the system function 'ERR()' will subsequently return the error number, and the square root is calculated from the expression:
   SQR(ABS(⟨expression⟩))

Type:

Statement

Purpose:

To stop execution of a program.

Syntax:

STOP

Execution:

Program execution stops and the following message is displayed on the screen:

STOP IN LINE nnnn

nnnn is the line number containing the 'STOP' statement.

Example:

```
540 STOP
```

Comments:

1. The 'STOP' statement is normally used to stop execution of a program other than at the end.

2. Program execution may be resumed by using the 'CON' command.

Type:

String function

Purpose:

To convert an arithmetic expression into a string.

Syntax:

STR$((expression))

Execution:

The arithmetic expression is converted to a string containing the characters which would be output if the value were printed by a 'PRINT' statement.

Example:

```
10 DIM B$ OF 7
20 INPUT "WRITE A NUMBER": A
30 B$ := STR$(A*1.5)
40 PRINT B$
```

Type:

Print function

Purpose:

To tabulate to the next character position in connection with a 'PRINT' statement.

Syntax:

TAB(⟨expression⟩)

Execution:

The arithmetic expression is evaluated and if necessary rounded. The result defines the start position of the next printout.

Example:

```
100 PRINT TAB(10),"THE RESULT IS: ",RESULT
```

Comments:

1. TAB(⟨expression⟩) can only be used in connection with 'PRINT' statements.

2. ⟨expression⟩ is an absolute value counted from the left hand margin of the output unit.

3. If the last printout before the 'TAB(⟨expression⟩)' has already passed the specified position, program execution is stopped with an error message.

4. The arithmetic ⟨expression⟩ must evaluate as greater than or equal to 1, and less than or equal to the maximum number of characters allowed in the width of the output device.

Type:

Trigonometric function

Purpose:

To calculate the tangent of an arithmetic expression.

Syntax:

TAN(⟨expression⟩)

Execution:

The tangent of ⟨expression⟩, given in radians, is calculated.

Example:

```
10 INPUT A
20 PRINT TAN(A)
```

Comments:

1. The arithmetic ⟨expression⟩ may be real or integer. The result will always be real.

Type:

Statement, command

Purpose:

To change the normal system response to a non-fatal error.

Syntax:

```
TRAP ERR-
TRAP ERR+
```

Execution:

During normal program execution any error will stop the program and will create an error message. However, a number of errors can be bypassed in a well-defined manner.
In these cases program interruption may be avoided by use of a 'TRAP ERR-' statement before the error arises. In this case, the system function 'ERR()' will return a value equal to the error number next time it is called (in all tests this will be considered true because it is not 0).
Program execution will then continue.

Example:

```
 10 INIT "", FILENAME$
 20 TRAP ERR-
 30 OPEN FILE 0, "XPLOCOMM", READ
 40 TRAP ERR+
 50 IF NOT ERR() THEN
 60  INPUT FILE 0: DEFAULT_FILENAME$
 70 ELSE
 80  DEFAULT_FILENAME$:="XPLOPROG"
 90 ENDIF
100 CLOSE
```

Comments:

1. Execution of a program starts by assigning the value false (= 0) to the system variable 'ERR()'. When a 'TRAP ERR-' statement has been executed, a non-fatal error assigns its error number to 'ERR()' which retains this value until its status is checked. Immediately after a such check, 'ERR()' is again assigned the value of false.
   Normally COMAL-80 sets a variable true by assigning it the value of 1, but here the error number is used.
   The error numbers are described further in appendix C.

2. After executing a 'TRAP ERR+' statement, the system returns to normal error handling.

3. During programming 'ERR' and 'ERR()' are interchangeable, but in program listings 'ERR()' is used.

Type:

Statement, command

Purpose:

To change the system response to the 'ESC' key.

Syntax:

```
TRAP ESC-
TRAP ESC+
```

Execution:

During normal program execution a check is made before each statement, to see whether the 'ESC' key has been pressed. If it has the program execution is stopped.
If a 'TRAP ESC-' statement has been executed, this function is blocked and the system function 'ESC()' will instead return the value of true (= 1) when 'ESC' is pressed.

Example:

```
10 TRAP ESC-
20 REPEAT
30  PRINT "THE 'ESC' KEY IS NOT PRESSED"
40 UNTIL ESC()
50 TRAP ESC+
60 PRINT "THE 'ESC' KEY WAS PRESSED"
```

Comments:

1. At the start of program execution, the system variable 'ESC()' is assigned the value of false (= 0). If a 'TRAP ESC-' statement is executed and the 'ESC' key is pressed after that, program execution continues but the system variable 'ESC()' is assigned the value of true (= 1) and retains this value until its status has been checked.
   Immediately after the value is used, 'ESC()' is again assigned the value of false (= 0).

2. The system returns to normal handling of the 'ESC' key after a 'TRAP ESC+' statement has been executed.

3. During programming 'ESC' and 'ESC()' are interchangeable, but in program listings 'ESC()' is used.

Type:

System constant

Purpose:

To assign the value of true to a boolean variable.

Syntax:

TRUE

Execution:

Returns the value 1.

Example:

```
 10 // PRIME
 20 //
 30 DIM FLAGS#(0:8190)
 40 SIZE1:=8190
 50 //
 60 COUNT:=0
 70 MAT FLAGS#:=TRUE
 80 //
 90 FOR I:=0 TO SIZE1 DO
100  IF FLAGS#(I) THEN
110   PRIME:=I+I+3
120   K:=I+PRIME
130   WHILE K<=SIZE1 DO
140    FLAGS#(K):=FALSE
150    K:+PRIME
160   ENDWHILE
170   COUNT:+1
180  ENDIF
190 NEXT I
200 PRINT "TOTAL NUMBER OF PRIMES: ",COUNT
```

Type:

Arithmetic function

Purpose:

To convert a real expression to an integer.

Syntax:

TRUNC((expression))

Execution:

The arithmetic (expression) is evaluated and the result is converted to integer type, decimals are disregarded.

Examples:

```
100 A=TRUNC(5.72)
100 A:=TRUNC(A/B)
```

Comments:

1. (expression) is real.
   The result is integer.

2. See also the 'ROUND' and 'INT' functions.

Type:

Command

Purpose:

To assign the background storage device which is to be the the default device.

Syntax:

UNIT (device)

Execution:

The internal pointers are updated to point at the stated device.

Examples:

```
100 UNIT "DK1:"
    UNIT DK1:
```

Comments:

1. (device) is defined as 2 letters describing the type of background storage device followed by the unit number and a colon.

Type:

String function

Purpose:

To convert a real number of string type to a number of real type.

Syntax:

VAL(<string expression>)

Execution:

The real number in <string expression> is converted to a number of real type.

Example:

```
10 DIM A$ OF 5
20 A$:="32.34"
30 PRINT VAL(A$)
```

Comments:

1. If <string expression> does not contain a correctly-formed real or integer number, program execution is stopped with an error message.

2. See the 'IVAL' function.

Type:

Machine code function

Purpose:

To find the absolute address in the memory at which a variable is stored.

Syntax:

VARPTR ((variable))

Execution:

The decimal, absolute address in memory at which the first byte of the variable (variable) is stored, is returned.

Example:

```
10 INPUT A
20 PRINT VARPTR(A)
```

Comments:

1. The result states where the first byte of the variable is stored. The remaining bytes are in the immediately following locations.
   Integers are stored in 2 bytes with the lower part of the number first.
   Real numbers are stored in 4 bytes in the 7-digit version.
   Real numbers are stored in 8 bytes in the 13-digit version.
   For string variables the first 2 bytes define the length and the string is then stored contiguously.

2. The result is of real type.

3. The variable may be an array with or without indices. If no indices are given, the address of the first element of the array is returned.

4. WARNING: In one situation a variable is moved after it has been allocated storage, thus changing its address. This happens, on exit from a non-closed procedure, to all variables encountered and allocated storage for the first time during the current call of the procedure.

Type:

Statement

Purpose:

To repeat the execution of a program section until the condition contained in the 'WHILE' statement is fulfilled.

Syntax:

```
WHILE (logical expression)
.
.
.
ENDWHILE
```

Execution:

On meeting the 'WHILE' statement the value of the (logical expression) is calculated. If this is true, execution resumes from the first executable statement following the 'WHILE' statement. When 'ENDWHILE' is reached execution continues with the 'WHILE' statement and the (logical expression) is evaluated again. If the (logical expression) is false the program continues from the first executable statement following the 'ENDWHILE' statement.

Example:

```
10 OPEN FILE 0,"DATA",READ
20 WHILE NOT EOF(0) DO
30  READ FILE 0: INDEX, NUMBER#, TEXT$
40 ENDWHILE
```

Type:

Statement

Purpose:

To write data in binary format to a data file.

Syntax:

WRITE FILE ⟨file No.⟩ [,⟨rec. No.⟩]: ⟨variable list⟩

Execution:

The values of the variables in ⟨variable list⟩ are written to the file contained in ⟨file No.⟩.

Examples:

```
100 WRITE FILE 7,REC_NO: A, B, C
100 WRITE FILE 3: A$, B#, C
```

Comments:

1. Before encountering a 'WRITE FILE' statement, a file must be opened and connection between ⟨file name⟩ and the ⟨file No.⟩ used in the 'WRITE FILE' statement must be established through the 'OPEN FILE' statement (or command), and type 'WRITE', 'RANDOM', or 'APPEND' must be established.

2. ⟨rec. No.⟩ is only used with 'RANDOM' files and is an arithmetic expression which will be rounded to integer if necessary.

3. ⟨file No.⟩ is an arithmetic expression.

4. ⟨variable list⟩ may contain all variable types. If an array variable is given without indices, the whole array will be written.

5. The elements in ⟨variable list⟩ are separated by commas.

6. During programming 'FILE' and '#' are interchangeable. In program listings 'FILE' is used.

Type:

System variable

Purpose:

To establish a new print-zone width by assigning this value to the system variable 'ZONE'.

Syntax:

ZONE:=<arithmetic expression>

Execution:

The system variable 'ZONE' is assigned the value of <arithmetic expression> which is rounded if necessary.

Examples:

```
100 ZONE:=8
100 ZONE=X*Y+3
    ZONE=12
    CURRENT:=ZONE
```

Comments:

1. On loading COMAL-80, 'ZONE' is assigned the value of 0.0. This value can only be changed by an assignment to 'ZONE'.

2. The 'NEW' command does not change the value of the system variable 'ZONE'.

3. See 'PRINT'

4. During programming ':=' and '=' are interchangeable. In program listings ':=' is used.