SPS/ACC/Note/85-33

30 January 1986
LEP Controls Note 63

NODAL REFERENCE MANUAL

P.D.V. van der Stok

TABLE OF CONTENTS

Section Page

CHAPTER 1

Introduction

1 Introduction

The NODAL language has been developed at CERN for the interactive control of the SPS accelerator. Its main design structure is based on FOCAL (command syntax and control structures) and SNOBOL (string handling). It allows an easy extension to different applications (not necessarily accelerator applications) through its extensible set of in-line functions.

A first description for the accelerator control of the SPS was done in "THE NODAL SYSTEM FOR THE SPS" - 1974, LAB 2-CO/74-2, December 1, 1974. An improved version of the language running on the NORD 100 computers has been published as a CERN yellow report and describes the NODAL language : M.C. Crowley-Milling and G.C. Shering, THE NODAL SYSTEM FOR THE SPS, CERN 78-07, sept 1978.

Until recently the NODAL language has been written in assembly language for different micro computers. To diminish the transport problems associated with the huge language environment it was decided to rewrite the interpreter in the MODULA-2 language. The opportunity was used to restructure the internal implementation of the language and to describe the syntax more rigorously. Consequently slight differences exist between the former NODAL implementations and the actual one described in this manual.

The in-line NODAL functions which constitute a target dependent addendum to the language are described elsewhere.

The syntax of the NODAL language is described in EBNF (Extended Backus Naur Formalism) notation. Below a description will be presented.

A construct is built up of a concatenation of syntactic factors i.e.

```
construct = factor1 factor2
```

On the other hand an alternative is presented as syntactic terms, i.e.

```
construct = term1 | term2
```

Parentheses () may be used to group terms and factors. If a construct is either a factor or nothing, the [] brackets are used. i.e.

```
construct = [factor1]
```

and when a construct includes a concatenation of a certain number (or none) of factors, the {} brackets are used :

```
construct = {factor}
```

Some examples are :

(F1|F2)(F3|F4)=(F1F3|F1F4|F2F3|F2F4)

F1[F2]F3 =(F1F2F3|F1F3)

F1{F2F3}=(F1|F1F2F3|F1F2F3F2F3|F1F2F3F2F3F2F3|.....

{F1|F2}F3 =(F3|F1F3|F2F3|F1F1F3|F1F2F3|F2F2F3|F2F1F3..

CHAPTER 2

Syntactic and program structures

2 Syntactic and program structures

The NODAL interpreter has two modes of operation :

- Interactive mode
- Program mode

The NODAL statements used in both modes are identical. However in one of the two modes the use of some of the statements are without meaning.

2.1 NODAL lines

syntax :

```
NODAL program line =
 line-digit [digit] . line-digit [digit] NODAL line
 line-digit = ('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')
 digit = ('0'| line-digit)
 NODAL line = NODAL statement { ';' NODAL line} |
                [ NODAL line ';'] '%' {character}
 NODAL statement = [ NODAL command <command syntax> |
                     <function call> ]
```

The interpreter knows about:
NODAL lines,
the NODAL program composed of NODAL lines,
NODAL defined functions composed of NODAL lines.

The NODAL lines in the program and the functions are preceded by a line number. A line number is composed of two parts :

- the group part
- the individual line part.

The parts are separated by a dot.

Both the group part and the individual part have a valid range of 1 to 99. A NODAL program or a NODAL defined function is thus composed of lines with the numbers 1.01 through 99.99. The line number where either the individual part or the group part is zero, is illegal. Line number 1.1 is equivalent to 1.10 and not equivalent to 1.01.

Valid numbers are : 1.01 99.01 55.55 40.40 1.99

Invalid numbers are : 0.0 0.1 44.0 100.1 1.997

The NODAL line consists of a set of NODAL statements separated by a semi colon (;). The empty statement is a NODAL statement with no characters or one or more blanks.

The exception is the % character which stands for comment, when placed at the beginning of a NODAL statement. All characters following this character on the same NODAL line are ignored by the interpreter. Also the semicolon character is then ignored by the interpreter.

Every NODAL statement starts with a NODAL command. The default (not explicitly stated statement) is the CALL command. A NODAL command may be followed by a set of literals and strings, the syntax of which depends on the command preceding them. The NODAL command is ended by the end of the line or by an end of command ';'. NODAL statements cannot be continued on to the next line.

2.2 Identifiers

The basic set of special characters of NODAL exists of :

+	addition sign
-	minus sign
*	multiplication sign
/	division sign
=	equal sign
,	comma separator
;	end of command
"	quote sign
'	alternative quote sign
<=	less equal sign
<>	unequal sign
<	less than sign or start concatenation in match
>	greater than sign; end concatenation in match
>=	greater equal sign
↑	exponentiation sign
:	error recovery point after match command
(	open bracket
)	close bracket
[	open square bracket or octal sign
]	close square bracket or octal format sign
[[	hexadecimal sign
]]	hexadecimal format sign
?	bit format sign
\	character value format sign
&	number format sign
$	string indirection sign or immediate assignment sign
.	deferred assignment sign
%	comment sign

Identifiers are names denoting variables and functions. An identifier may contain up to six characters. The first character should be a letter. This may be followed by other letters, numbers, dot or colon.

syntax :

```
NODAL name = capital { capital | digit | : | . }
```

Valid names are :

```
AB    BBAB    R    R12345    RT    RT....    RT:::A
```

Non-valid names are :

```
BA;    BA-    B1234567
```

The command names to be found at the beginning of a NODAL statement :

A(SK)	C(ALL)	DI(MENS)	DE(FINE)
DO	ED(IT)	EN(D)	ER(ASE)
EX(ECUT)	F(OR)	G(OTO)	IF
IM(EX)	LI(ST)	LO(AD)	OL(D)
OP(EN)	OV(ERLA)	Q(UIT)	REM(IT)
RET(URN)	RO(F)	RU(N)	SA(VE)
SE(T)	T(YPE)	V(ALUE)	WA(IT)
WH(ILE)	$A(SK)	$D(O)	$I(F)
$M(ATCH)	$P(ATTE)	$S(ET)	$V(ALUE)
?ON	?OF(F)		

syntax :

```
NODAL command = askcom | callcom | dimcom | defcom | docom |
                editcom | endcom | erasecom | execcom |
                forcom | gotocom | ifcom | imexcom | listcom
                | loadcom | oldcom | opencom | overcom |
                quitcom | remitcom | returncom | rofcom |
                runcom | savecom | setcom | typecom |
                valuecom | waitcom | whilecom | dollarask |
                dollardo | dollarif | dollarmatch |
                dollarpat | dolarset | dollarvalue | oncom |
                offcom

askcom       = 'A' | 'AS' | 'ASK'
callcom      = 'C' | 'CA' | 'CAL' | 'CALL'
dimcom       = 'DI' | 'DIM' | 'DIME' | 'DIMEN' | 'DIMENS'
defcom       = 'DE' | 'DEF' | 'DEFI' | 'DEFIN' | 'DEFINE'
docom        = 'DO'
editcom      = 'ED' | 'EDI' | 'EDIT'
endcom       = 'EN' | 'END'
erasecom     = 'ER' | 'ERA' | 'ERAS' | 'ERASE'
execcom      = 'EX' | 'EXE' | 'EXEC' | 'EXECU' | 'EXECUT'
forcom       = 'F' | 'FO' | 'FOR'
gotocom      = 'G' | 'GO' | 'GOT' | GOTO'
ifcom        = 'IF'
imexcom      = 'IM' | 'IME' | 'IMEX'
listcom      = 'LI' | 'LIS' | 'LIST'
loadcom      = 'LO' | 'LOA' | 'LOAD'
oldcom       = 'OL' | 'OLD'
opencom      = 'OP' | 'OPE' | 'OPEN'
overcom      = 'OV' | 'OVE' | 'OVER' | 'OVERL' | 'OVERLA'
quitcom      = 'Q' | 'QU' | 'QUI' | 'QUIT'
remitcom     = 'REM' | 'REMI' | 'REMIT'
```

```
returncom     = 'RET'|'RETU'|'RETUR'|'RETURN'
rofcom        = 'RO'|'ROF'
runcom        = 'RU'|'RUN'
savecom       = 'SA'|'SAV'|'SAVE'
setcom        = 'SE'|'SET'
typecom       = 'T'|'TY'|'TYP'|'TYPE'
valuecom      = 'V'|'VA'|'VAL'|VALU'|'VALUE'
waitcom       = 'WA'|'WAI'|'WAIT'
whilecom      = 'WH'|'WHI'|'WHIL'|'WHILE'
dollarask     = '$A'|'$AS'|'$ASK'
dollardo      = '$D'|'$DO'
dollarif      = '$I'|'$IF'
dollarmatch   = '$M'|'$MA'|'$MAT'|'$MATC'|'$MATCH'
dollarpat     = '$P'|'$PA'|'$PAT'|'$PATT'|'$PATTE'
dollarset     = '$S'|'$SE'|'$SET'
dollarvalue   = '$V'|'$VA'|'$VAL'|'$VALU'|'$VALUE'
oncom         = '?ON'
offcom        = '?OF'|'?OFF'
```

All commands can be shortened to their minimum number of characters. The above tables show the shortest possible form of the command and all larger equivalents.

It should be noted that the command names are not reserved words. It is only at the beginning of a statement that the above mentioned names have a special meaning. E.g.

```
SET SET = 1;
```

is a legal statement. The first SET is a command name while the second SET is a variable name.

2.3 Numbers

syntax :

```
number      = ['+'|'-'] integer |  real | hexa | octal
integer     = digit { digit }
real        = digit { digit } '.' {digit} [ scalefactor ]
scalefactor = 'E' ['+'|'-'] digit {digit}
octal       = '[' octal-digit { octal-digit}
octal-digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'
hexa        = '[[' hexa-digit { hexa-digit }
hexa-digit  = digit|'A'|'B'|'C'|'D'|'E'|'F'
```

Decimal notation is used for numbers. The letter E is used for exponential notation with base 10. Correct numbers are :

```
1      12345          12.34       123.45E-2     -1234.78E+08
```

Octal representation is denoted by preceding the number with the [sign. Hexadecimal numbers are preceded by a [[sign.

Valid numbers are :

[1234 [0 [[FFFF [[A6

Uncorrect numbers are :

1,1 1e234 1EXP4 1E4/5
[A8 [98 [[GG

2.4 Strings

syntax :

string = "'"{ character }"'"|"'" { character } "'"

Sequences of characters enclosed by single quotes or by double quotes are called strings. Examples of strings are :

"assg4$$55" '778ghhhh' 'strin"' '"' "'" '7;8'

Invalid strings are :

ddddd" ff'fff' "ddddd' "ddd"'

CHAPTER 3

Data structures

3 Data structures

Data types can be specified anywhere during program execution. Data types can be modified during execution. Simple variables are automatically created by the assignment statement : SET. Composite data types are created by the

command DIMENS

There are three basic data types :
- type REAL
- type INTEGER (only in arrays)
- type STRING

A set of operations can be executed on each of these types. When an arithmetic operation is executed the type INTEGER is always converted into REAL, before the operation is executed.

3.1 Simple variables

Any simple variable is either a STRING or a REAL. Values of type REAL are real numbers. The available basic operations are :

+	add
-	minus
*	multiply
/	divide
↑	exponentiate

It is important to remember that REAL values are stored as a number sequence and an exponent. The maximum and minimum numbers which can be stored depend on the size of the exponent and the number sequence, while the accuracy of the numbers is determined by the number of characters in the number sequence (floating point representation). The values are stored internally as 64 bit IEEE floating point format numbers and they are stored on files and transported as 32 bit IEEE floating point format numbers.

The type STRING is represented as a sequence of characters with a maximum length of 80 characters. Each string has a size allocated to it, which denotes the total number of characters present in the string. No end of string character is available. No quotes are stored at the beginning and end of string.

3.2 DIMENS command

syntax :

```
DIMENS statement = dimcom ['-' int |'-' str] identifier
                   '('expression [',' expression]')'
 int = 'I'|'IN'|'INT'|'INTE'|'INTEG'|'INTEGE'|'INTEGER'
 str = 'S'|'ST'|'STR'|'STRI'|'STRIN'|'STRING'
```

Composite types are created with the DIMENS command.

DIM-I creates INTEGER arrays.
DIM creates REAL arrays.
DIM-S creates STRING arrays.

The statement DIM NAME erases first the variable NAME if it already exists, and then creates a new variable of the same name NAME. This is true for all three types of DIMENS commands mentioned above.

3.2.1 Real arrays

The command DIM A(N) creates an array A with N floating point numbers. The numbers are stored contiguously in memory. After creation their position does not change until they are erased, redefined or garbage collection takes place. An alternative call is : DIM A(N,M) which creates a two dimensional array A.

The two dimensional arrays are stored by column, i.e. the first subscript varies most rapidly. The above array is stored as :

```
A(1,1)  A(2,1)  A(3,1) ..........      A(N,1)
A(1,2)  A(2,2)  A(3,2) ..........      A(N,2)
  .                                      .
  .                                      .
  .                                      .
A(1,M)  A(2,M)  A(3,M) ..........      A(N,M)
```

The two dimensional array can also be accessed as a one dimensional one. The access to A(x,y) is then equivalent to accessing A(x + (y-1)*N)

N.B. A(k,1) is equivalent to A(k), even for one dimensional arrays.

3.2.2 Integer arrays

Integer arrays can be defined with the DIM-I command. They are stored as 16 bit 2's-complement. Both one dimensional and two dimensional arrays are allowed. The storage order is as described above for REAL arrays.

3.2.3 String arrays

Apart from the string variables also one dimensional string arrays can be defined. They have no predefined number of elements.The individual members are character strings with a maximum of 80 characters per member. The characters of each individual member are stored contiguously, while the members themselves are not stored in a predefined fashion. Actually any member can at a given moment be anywhere in memory. Once defined a member of a STRING array is only moved when a new assignment is done to it. The arrays are defined with the command DIM-S. After the definition of the string array, the individual members are defined by assigning strings to them. It is possible to first define element n and afterwards element m where m may be smaller or larger than n and does not need to be adjacent to n.

String array elements to which no values have been assigned, will return an empty string.

Having defined A as string array first, it can afterwards be redefined as a REAL array by invoking the command DIM. However to redefine A as a simple variable, one first needs to ERASE A and then redefine A by doing an assignment to A, which will then automatically create the simple variable A. Once A is defined as an array or simple variable it should be used as such in assignments, expression or concatenations.

3.3 ERASE command

syntax :

ERASE statement = erasecom { identifier | group | line }

The erase command serves to erase NODAL defined functions, variables, lines or complete groups. As many identifiers, group numbers or line numbers may be specified as can be written on one NODAL line.

It is not allowed to erase inline-NODAL functions.

The use of the ERASE command for program lines is discouraged in program mode. It is always possible to erase lines which are currently active, thus creating situations which have unpredictable consequences.

Some standard uses of ERASE can be made with standard identifiers :

```
ER ALLD means ERASE all NODAL defined functions.
ER ALLP means ERASE all NODAL program lines.
ER ALLV means ERASE all NODAL variables.
ER ALL  means ERASE all NODAL variables and program lines
ER      is equivalent to ER ALLP
```

N.B. When the ERASE command is used in program mode, great care should be exercised not to erase the lines which are currently active. In that case the behaviour of the interpreter is unpredictable. There is only one instance where the current line may be erased or the whole program may be erased that is in a $DO command followed by a RUN :

```
10.10 $DO "ERASE ALL; RUN file"
```

CHAPTER 4

Assignments

4 Assignments

4.1 SET command

syntax :

SET statement = setcom identifier'=' expression

An assignment statement starts with a SET command, followed by an identifier, a '=' and finally an arithmetic expression.

The meaning of this command is :

- Find the identifier; when non existent , create it.
- Evaluate the expression, which yields a value.
- Feed the obtained value into the identifier.

The identifier can be :

- A simple variable.
- A REAL or integer array element.
- A write or read/write in-line function.
- A NODAL defined function

The identifier cannot be a string, a string element or a string function.

For example with A a simple variable, B a two dimensional array and FUNC a write in-line function, the following is allowed :

```
SET A = expression
SET B(n,m) = expression
SET FUNC(param) = expression
```

Not allowed is:

```
SET B = expression
SET A(1,3) = expression
```

When C does not exist, the expression SET C = expression is allowed and the variable C will be created. However, when C does not exist the statement SET C(m) = expression is not allowed and C is not created.

When B is an integer array the result of the expresssion is truncated to the highest possible positive value (7FFF hexadecimal) and then stored in the array element of B.

4.2 Expressions

syntax:

```
Expression = ["-"|"+"] operand { operator operand}
operator   = "-"|"+"|"/"|"*"|"↑"
operand    = '(' Expression ')' | number | identifier
```

An expression is composed of operands and operators. It is evaluated by applying the operands on the operators in the correct order. The operators may be :

- addition (+)
- subtraction (-)
- multiplication (*)
- division (/)
- exponentiation (↑)

The operands may be :

- Simple variables.
- Array elements (INTEGER or REAL).
- Read or read/write in-line functions.
- NODAL defined functions.

Parts of expressions may be enclosed in parentheses. The part of the expression contained within the parentheses should be an expression in its own right. The following additional rules apply :

- Every variable in an expression should exist.
- Two operators must never be written side by side.
- Two operands must never be written side by side.
- Precedence rules determine the order of evaluation.
- Parentheses can be used to force precedence.

The order of evaluation is from left to right. The order of precedence of the operators is first exponentiation, then multiplication, then division, then subtraction and finally addition. Expressions enclosed in parentheses are evaluated first. Parentheses can be nested. In that case first the expressions of lower level are evaluated before the higher level expression can be evaluated.

Valid expressions are :

```
2*3 + 4*5              yields 26
10↑3*3/10 + 21 - 2     yields 319
6 + 6 * 2              yields 18
(6 + 6)*2              yields 24
1/2*5*4                yields 0.025
(1/2)*5*4              yields 10
```

C H A P T E R 5

Control structures

5 Control structures

It is of prime importance that statements can be selected and/or executed repetitively dependent on conditions specified in the program. Hence the sequence of actions which are executed in a program depends on the data of the program. The paths through the program are specified by selecting the NODAL lines or NODAL groups which have to be executed. So most NODAL control statements either act on the statements following the condition on the same NODAL line or specify the NODAL group or line to which the program path has to be deferred. Below the individual control statements will be discussed.

5.1 DO command

Syntax :

DO statement = docom expression {'!' expression} ['!']

The DO command specifies the group or line which has to be executed. After execution the control returns to the statement following the DO command. The DO command is the most widely used command in combination with other conditional commands. As most conditional commands only allow the execution of the two or three NODAL statements following them on the same line, the number of executable statements can be enormously enlarged with the DO command.

The following example will show a possible use of the DO command.

```
1.1 DO 10; DO 20; END

10.10 NODAL statements
10.20 NODAL statements

20.10 NODAL statements
```

Executions starts at line 1.1. After the first DO command the lines 10.10 and 10.20 are executed. Hereafter control returns to the statement following the DO 10 statement : DO 20. Now line 20.10 is executed, control returns to line 1.1 and the program ends with the END statement.

Both the interactive use and the program use of DO is allowed.

After evaluation of the expression following the DO command the obtained number is rounded to the second decimal. When the number has no decimal part a DO group is executed. In the other case only the specified line is executed.

When the specified line or group does not exist, an error is generated.

The DO command allows to go from interactive mode to program mode. The DO command in an interactive NODAL line starts the execution of the specified group or line. Together with the RUN command, this is the only way to go from interactive to program execution mode.

5.1.1 Exceptions

The exclamation mark "!" in the DO command syntax is used to specify actions after the occurrence of an error. Normally when an error occurs in a NODAL program the execution is terminated and the error is displayed on the screen. However it is not always advisable to stop the execution of the program when an error is detected. It may be the main purpose of the program to wait for errors and then execute a specially designed error sequence. To cater for this the ! sign in the DO command specifies the group or line to be executed after the occurrence of an error in the formerly executed line. See example below.

```
 1.1  DO 10!20; END

10.10 DO 50

20.10 Error display

50.10 Provoke error
```

At line 1.1 group 10 is executed which executes group 50, which provokes an error. Control then returns to the first DO command and now group 20 is executed. Control then returns to the statement after the DO statement in line 1.1 and the programs ends.

Error returns can be multiple or omitted :

DO n! means execute group n and return to the statement following the command, independent of the correct execution of n.

DO k!l!m!n means execute k, if error execute l, if error execute m, if error execute n, if error pass error to next DO command with exclamation mark. When no error, execute the statement following the DO statement.

The above example shows that the DO commands and the exclamation marks are nestable.

```
 1.1  DO 10!20; END

10.10 DO 30!40

20.10 Display error

30.1  provoke error 1

40.1  provoke error 2
```

In this example the error at line 30.1 is trapped at line 10.10. However the error in 40 is trapped at line 1.1. At line 20 error 2 (the last one) is then displayed.

5.1.2 Recursion

Recursive use of the DO command is also possible. The expression DO n in group n is allowed and correctly executed till any depth dependent on the stack space available to the program. However the programmer should be aware that all group n actions are executed on the same local variables. If one wants to write recursive procedures it is recommended to use the NODAL defined functions.

5.2 END command

syntax :

END statement = endcom

The END command terminates the execution of a NODAL program or NODAL defined function. When the END is met in a program the interpreter returns to interactive mode or the interpreter is stopped. When the END is executed in a NODAL defined function control is returned to the NODAL line invoking this function.

5.3 RETURN command

syntax :

RETURN statement = returncom

The execution of a DO group can be terminated by invoking the RETURN command. Its action is :
Terminate the execution of the current do group or line and return to the command following the invoking DO command. When no DO command was executed the program is terminated. In the latter case the RETURN is equivalent to an END command.

5.4 GOTO command

syntax :

GOTO statement = gotocom expression

The GOTO command serves to tell the interpreter that execution should continue at another line. Its action is :

- Evaluate expression.
- Transfer execution to the line specified by expression.

The interpreter does not keep track of the line from which the GOTO command was invoked. The control transfer depends however on the DO command. If the line specified in the GOTO command lies outside the group specified by the currently active DO command, then a RETURN is executed and control returns to the statement following the invoking DO command. When the line containing the GOTO is called by a DO line statement then all GOTO statements will provoke a RET command, unless a GOTO its proper line number is specified. For example :

```
 1.1   DO 10; END

10.1   GOTO 10.3
10.2   NODAL statements
10.3   GOTO 12.4
10.4   NODAL statements

12.4   NODAL statements
```

In this example control goes to line 10.1 then to 10.3 and back to 1.1, because 12.4 lies outside group 10. If 1.1 is replaced by DO 10.1, then only line 10.1 is executed before control returns to line 1.1.

The GOTO statement placed after a WHILE or FOR command allows an exit from the repetitive loops.

N.B. the GOTO statement is only executed when in program mode. Consequently a GOTO statement in an interactive line is equal to a return to the interactive command mode of the interpreter.

5.5 WHILE command

syntax :

```
WHILE statement =
        whilecom expression relation expression
          { 'OR' expression relation expression }

relation = '<' | '<=' | '<>' | '=' | '>' | '>='
```

The sequence of actions of the WHILE command are :

- Evaluate first expression.
- Evaluate second expression.
- Compare values.
- Compare comparison result with condition.
- Execute rest of line when comparison fits condition.
- Search for OR, retest condition and execute if true.
- At end of line return to WHILE.

The WHILE statement serves to execute a certain mumber of times (as long as a certain condition holds) the statement sequence following the WHILE command on the same line.

Two expression are supposed to be equal when they differ less than 5E-8.

For example the statement :

```
WHILE 0=0; NODAL statements
```

will indefinitely execute the NODAL statements following the WHILE command.The statement :

```
WHILE A > 1 OR A <= 1; NODAL statements
```

will also be executed indefinitely because one of the two statements is always true.

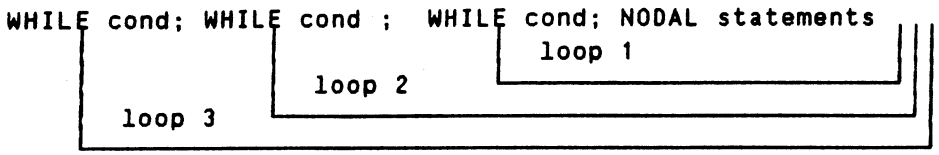
However the NODAL statements below :

```
WHILE A > 1; WHILE A < 1; NODAL statements
```

will never be executed, because one of the two statements is false. When A is smaller than 1 the first WHILE statement will be executed only once. When A is bigger than 1 the second WHILE statement will be executed indefinitely because A will stay larger than 1 and the first condition is eternally fulfilled.

From above it can be deduced that WHILE statements can be nested on the same line.

The order of nesting is shown in the diagram below.



The outer loop always encloses the inner loop. When all three conditions are true the inner loop 1 is executed until its condition is false. Control then returns to loop 2 and back to loop 1, etc until all three conditions are false.

When a GOTO statement is placed after a WHILE statement, an exit from the WHILE loop is provided. Actually a GOTO inside a WHILE loop invoked by a DO comand is equivalent to a RETURN command when the GOTO is outside the group.

Example

```
    1.1   DO 2; END

    2.1   WHILE cond; GOTO 3.1
    2.2   NODAL statements

    3.1   NODAL statements
```

is in this context equivalent to :

```
    1.1   DO 2; END

    2.1   WHILE cond; RETURN
    2.2   NODAL statements

    3.1   NODAL statements
```

5.6 FOR command

syntax:

FORstatement = forcom
identifier'=' expression [, expression] [, expression]

The sequence of actions for the FOR command is :

- Find identifier, when non existent create it.
- Evaluate expression 1.
- Evaluate expressions 2 and 3 when available.
- Feed the value of expression 1 into control variable.
- Store result of control variable into identifier.
- Execute NODAL commands on same line till end of line.
- Return to FOR command and increment control variable.
- Store result of control variable into identifier.
- When maximum value obtained leave FOR loop.

The FOR command allows the repetitive execution of the NODAL statements on the same line following the FOR command. The semantic meaning of the parameters are :

- first expression(compulsory) : start value of identifier.
- last expression (optional) : end value of identifier.
- middle expression (optional) : increment of identifier.

Default value of last expression is first expression value . Default value for middle expression is 1.

The statement :

FOR I = expression ; NODAL statements

is equivalent to :

SET I = expression ; NODAL statements

The expression

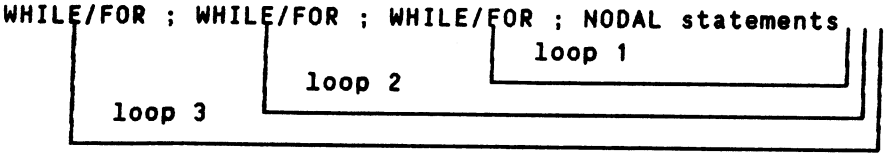
FOR I = expression, expression; NODAL statements

is executed as:

SET I to first expression, and execute following statements. At end of line increment I by 1, if I is larger than the second expression, exit from FOR loop and execute the following line. If a third expression is added, I is incremented by the value specified by the middle expression.

N.B. The expressions in the FOR statements are evaluated only once at the beginning of the execution, and the end value, increment and start value are unchanged, even when a parameter of the expressions is changed within the FOR loop statements.

FOR loops can be nested as the WHILE statements. FOR loops can be nested inside WHILE loops and vice versa as shown in the diagram below



The outer loop always encloses the inner loop. When all three conditions are true the inner loop 1 is executed until its condition is false. Control then returns to loop 2 and back to loop 1, etc until all three conditions are false. Conditions are either the WHILE conditions or the verification of the upper value of the FOR control parameter.

When a GOTO statement is placed after a FOR command, an exit from the FOR loop is provided. Actually a GOTO inside a FOR loop invoked by a DO command is equivalent to a RETURN command when the GOTO is outside the group.

5.7 ROF command

syntax :

ROF statement = rofcom

The purpose of the ROF command is to terminate the current FOR loop. The ROF applies to the last active FOR statement even if several DO statements are currently active between the FOR command and the ROF command.

Example :

```
  1.1    FOR I = 1,10; DO 10
  1.2    END

 10.1    FOR K = 1,10; DO 20

 20.1    NODAL statement
 20.2    DO 30

 30.1    NODAL statement
 30.2    ROF
```

Every time ROF is met control returns to the NODAL line following line 10.1. This implies a return to the FOR statement in line 1.1. Consequently the first FOR statement is executed for all 10 values of I, and the second FOR statement is executed 10 times for one value of K.

5.8 IF command

There are two types of IF commands.

syntax :

IF statement = logical IF | arithmetic IF

logical IF = ifcom expression relation expression
{ 'OR' expression relation expression }

arithmetic IF = ifcom '(' expression ')'
expression [',' expression] [',' expression]

The sequence of actions of the logical IF command are :

- Evaluate expression.
- Evaluate second expression.
- Compare values.
- Compare comparison result with condition.
- Execute rest of line when comparison fits condition.

- Search for OR, retest condition and execute if true.

The sequence of actions for the arithmetic IF are :

- Evaluate expression.
- Test sign of result.
- If sign negative, execute a GOTO to the line specified by first expression.
- When expression = 0, execute a GOTO to the line specified by second expression or continue NODAL line.
- When expression > 0, execute a GOTO to the line specified by the third expression or continue NODAL line.

The expressions of the logical IF are equal when their absolute difference is less than | 5E-8 |. The expression within brackets of the arithmetic IF is zero when its absolute value is less than | 1E-6 |. Some examples will be shown below :

```
IF  A=1 OR A=3; SET A = A+1
```

When A has the value 1 or 3 the value of A will be incremented by 1

```
IF A=1; IF A=3; SET A = A+1
```

The condition expressed above means if A has the value 1 and the value 3 simultaneously, then increment A. As a consequence A is never incremented. By placing several IF statements in cascade, an AND of conditions is obtained.

The arithmetic IF statement can be used as shown in the examples below :

```
1.1  IF (X) 10.1; NODAL statements
```

When X < 0 then control is tranferred to line 10.1. In all other cases the NODAL statements following the IF statement are executed.

```
1.1  IF (X) 10.1, 10.2; NODAL statements
```

When X < 0 then control is transferred to line 10.1. If X = 0 then control is transferred to line 10.2. In all other cases the NODAL statements following the IF statement are executed.

```
1.1  IF (X) 10.1, 10.2, 10.3; NODAL statements
```

In this case the NODAL statements following the IF statement are never executed. When X > 0 then control is transferred to line 10.3.

N.B. As the GOTO statement has no sense in interactive mode, the GOTO of the arithmetic IF statement is never executed when in interactive mode.

CHAPTER 6

String assignments

6 String assignments

6.1 $SET command

syntax :

$SET statement = dollarset identifier '=' concatenation

The meaning of this command is :

- Find the identifier, when not existent create it.
- Evaluate the concatenation.
- Feed the obtained string into the identifier.

The identifier can be :

- A simple string variable.
- A string array element.
- A write or read/write in line string function.
- A NODAL string defined function.

For example with S1 a simple string variable, S2 a string array and SF a write in-line string function, the following is allowed :

```
$SET S1        = concatenation
$SET S2(n)     = concatenation
$SET SF(param) = concatenation
```

Not allowed is :

```
$SET S2       = concatenation
$SET S2(n, m) = concatenation
$SET S1(n)    = concatenation
```

When SV does not exist, the statement $SET SV = concatenation is allowed and the string variable SV will be created. However when SV does not exist the statement $SET SV(n) = concatenation is not allowed and SV is not created.

The total length of the result string of the concatenation may not exceed 80 characters.

6.2 Concatenations

A concatenation is composed of string elements and expressions which may be preceded by a format definition.

syntax :

```
concatenation = (string | [format] expression |
                        identifier | control) [concatenation]
format  = '%' expression | '%,' | ']' | ']]' | '?'
control = '!' | '\' expression | '&' expression
```

A string may be :

- A literal string.
- String variable.
- String array element.
- Read or read/write in-line string functions.
- NODAL string functions.

A control may be :

- single character value preceded by \ (\ expression).
- CR LF (!).
- spaces (& expression)

The identifier may be :

- String variable.
- String array element.
- Read or read/write in-line string functions.
- NODAL string defined functions.

A format is :

```
% expression : change format to field length specified by
               expression.
%,           : return value in exponential representation.
]            : return value in octal representation.
]]           : return value in  hexadecimal representation.
?            : return value in binary representation.
```

The newly defined format field with the '%' character is valid for all numbers following the format specification within the same NODAL statement.

The field length definition of a format is of the form n.m. N designates the total field length, while m*100 expresses the number of digits behind the decimal point. The default value is 11.04 meaning 4 digits behind the decimal point and 6 digits or blanks before the decimal point.

The total field length should be smaller than 80.

When the number m is zero (no fractional part) the number will be visualised as a right adjusted integer with total field length n.

The exponential format can be forced by using '%,' and is automatic when the number exceeds the space allocated by the current format specification.

The exponential notation will visualise the number with a total field length of 12. The last part is the value of the exponent preceded by the E digit. The first part is the decimal number with one digit for the integer part followed by the decimal point and the fractional part. The number one will be displayed as :

```
1.0000000E0
```

The octal representation is preceded by one blank and followed by 5 octal digits.

The hexadecimal representation is preceded by one blank and followed by four hexadecimal digits.

The binary representation consists of 16 binary digits.

The insertion of an expression into a concatenation is done in the following way.

- Evaluate expression.
- Write result of expression into a string dependent on format directive.
- Concatenate string to result string.

Some examples are :

```
$SET ST = "HELP" "ME"
results in :
HELPME

$SET ST = "HELP" ! "ME"
will result in :
HELP
ME
```

The latter is equivalent to:

```
$SET ST = "HELP" \10 \13 "ME"
```

The expression:

```
$SET ST = "HELP" ! ]]32 "ME"
will result in :
HELP
 0040ME

$SET ST = "HELP" &8 "ME"
will result in :
HELP        ME
```

N.B It should be noted that the total maximum length of a concatenation is 80 characters.

6.3 Identifier indirection

syntax :

identifier = {'$'} NODAL name

The left hand identifier may be any identifier as specified in the syntax of the individual commands. The right hand NODAL name, when preceded by a '$', is the name of :

- a string variable.
- string array.
- NODAL string in-line function.
- NODAL string defined function.

The action is :

Take the last dollar sign in the chain of dollars and replace it together with the name of the identifier by the contents of the identifier. Use the thus obtained identifier as identifier. When another dollar precedes the newly acquired identifier, repeat the sequence. This also means that dollars contained in the identifier preceded by a dollar are added to the sequence of indirection dollars.

Below some examples will follow :

```
$SET A="B"; $SET C=$A
```

will put the contents of B into C as $A is replaced with B and the last statement becomes $SET C=B.

```
$SET A="$B"; $SET B="$A" ;$SET C=$A
```

will result in an eternal loop as the $A will be replaced by a $B and the $B by a $A ad infinitum. The loop will terminate with a stack overflow.

```
$SET C="B"; $SET B="D"; $SET A=$$C
```

will result in the contents of D to go to A, as $$C will be replaced with D. This is equivalent to :

```
$SET C="$B"; $SET B="D"; $SET A=$C
```

The name indirection can be used for every identifier. The following actions are completely valid.

```
$SET XX="B"; DIM $XX(10); FOR I =1,10; SET $XX(I)=I
```

The array B of dimension 10 is created and the contents of its members is set to their array indices.

CHAPTER 7

String control structures

7 String control structures

Apart from the control structures used to take decisions on the results of expressions, two other string based structures are available. The two string based possibilities are:

- Execute a concatenation.
- The string equivalent of the IF command.

7.1 $DO command

syntax :

$DO statement = dollardo concatenation

The actions associated with this command are :

- Create the result string of the concatenation
- Execute this string by the interpreter.

Some examples are

```
$SET ST = "SET A=1"; $DO ST
```

will set the variable A to 1. To create the variables A to Z and assign values to them the following can be tried.

```
FOR I =65,90;$SET ST="SE "\I"=I";$DO ST
```

The same tricks can be done with arrays. Suppose the array ST to be prepared with a set of NODAL lines. Then the following statement will execute them all.

```
FOR I=1,N; $DO ST(I)
```

7.2 $IF command

Again there are two types of IF commands: the logical $IF and the arithmetic $IF command. The logical $IF is :

syntax :

$IF statement = logical $IF | arithmetic $IF

logical $IF = dollarif concatenation relation concatenation
{ 'OR' concatenation relation concatenation }

```
arithmetic $IF =
 dollarif '('identifier {identifier} '-' concatenation ')'
            expression[',' expression ] [',' expression ]
```

identifier may be :

- String variable
- String array element
- NODAL string function
- NODAL in-line read/write string function

The sequence of actions for the logical $IF command are :

- Store result string of concatenation
- Store second result string of concatenation
- Compare the two strings lexically
- Compare comparison result with condition
- Execute rest of line when comparison fits condition
- Search for OR, retest condition and execute if true

The sequence of actions for the arithmetic $IF are :

- Concatenate string identifiers until minus sign.
- Store result string.
- Store second result string of concatenation.
- Compare the two strings lexically.
- IF first string is smaller execute a GOTO to the line specified by the first expression.
- IF strings are equal execute a GOTO to the line specified by the second expression or continue NODAL line.
- IF first string is larger execute a GOTO to the line specified by the third expression or continue NODAL line.

The lexical comparison is done on a character by character basis where the value is determined by the ASCII values of the characters, where the following rule holds :

NULL<0<9<A<Z

Two strings are different from the first character position on which the two character values differ, independent of the size of the strings. When two strings are the same but one string is longer than the other, then the longer has the largest value.

Some examples will be shown below :

```
$IF A="YES" OR A="NO"; NODAL statements
```

The NODAL statements following the $IF command will be executed when A contains either YES or NO.

```
$IF A="YES"; $IF A="NO"; NODAL statements
```

In this case the NODAL statements will never be executed because A has to contain YES and NO simultaneously for the two conditions to be true. An effective AND of string comparisons is obtained by a sequence of $IF statements.

The arithmetic $IF can be used as shown below :

```
$IF (A - "YES") 10.1, 10.2 ,10.3
```

The contents of the variable A is compared with YES. If the lexical contents is smaller a GOTO 10.1 is executed; when it is equal or larger a GOTO 10.2 and 10.3 is executed respectively.

N.B. As the GOTO statement has no sense in interactive mode, the GOTO of the arithmetic $IF statement is never executed when in interactive mode.

CHAPTER 8

Pattern and Match structures

8 Pattern and Match structures

The $MATCH command makes use of patterns which can be defined with the $PATTE command. This part of the NODAL language is almost an integral copy of the SNOBOL4 language.

8.1 $PATTE command

syntax :

```
$PATTE statement = dollarpat identifier "=" pattern
 pattern = simplepattern ['.'NODAL name |'$'NODAL name]
          [ pattern ]
 simplepattern = (NODAL name|string) {NODAL name|string} |
                 '(' pattern ')' |NODAL name'!'NODAL name
```

A NODAL name can be :

- simple string variable.
- string array item.
- inline string function.
- inline pattern function.
- NODAL string defined function.

The $PATTE command takes the pattern defined on the right hand side of the equal sign and stores the encoded result into the pattern variable specified on the left hand side.

If the variable does not exist a pattern variable is created. If the variable exists but is not a pattern variable then an error is generated.

The result of a pattern is a string. The pattern functions operate on the string and determine which part of the string should be selected when a match is being done. It can position at a special place in the string or ignore a set of characters in the string.

Alternation of strings is defined by the ! character and grouping is specified by the '(' and ')' brackets.

For example the pattern :

```
"YES" ! "NO"
```

results in the strings "YES" and "NO". This can also be expressed by :

```
"Y" "E" "S" ! "N" "O"
```

The example with brackets shows its function in connection with the ! character :

```
"A" ( "B" ! "C" ) "D"
```

results in the strings "ABD" and "ACD". The brackets denote the construction of all possible strings enclosed in the brackets and to add to those results, all the possible string combinations which surround the brackets.

The additional facility of storing result strings into string variables or string functions is connected with the $MATCH command and will be treated in the next section. The result string is denoted by :

```
.NODAL name
$NODAL name
```

NODAL name stands for :

- string variable.
- string array element.
- read/ write inline string functions.
- write only in-line string functions.
- NODAL string defined functions.

The precedence rules for a pattern construction are : Construction from left to right with precedence for the operators in the order shown below.

- patterns contained in brackets ()
- direct assignment or deferrred assignment .$
- sequences of patterns
- alternative possibilities !

The pattern

```
"Y" "E" "S" ! "N" "O"
```

is equivalent to

```
( "Y" "E" "S" ) ! ( "N" "O" )
```

The pattern

```
"Y" "E" ( "S" ! "N" ) "O"
```

means :

YENO and YESO

The pattern

```
P1 P2 .A1 P3 P4 $ A2
```

is equivalent to

```
P1 ( P2 .A1) P3 ( P4 $ A2)
```

This means that A1 will only contain the partial match of P2 and A2 only the partial match of P4. The time of storage of the result in A1 (at the end of a complete match of the total pattern defined above) has nothing to do with the

precedence of the pattern operations defined in the pattern above.

Another example is :

```
P1 .A1 ! P2 .A2
```

is equivalent to

```
( P1 .A1) ! ( P2 .A2)
```

which means that after a match of P1, assignment to A1 will take place and the matching of P2 will not take place and no assignment to A2 will happen. Assignment to A2 will only happen when P1 fails (no assignment to A1) and P2 succeeds.

The pattern functions are treated separately in the NODAL function manual and in the section 8.3. During the matching of a pattern to a string, these functions execute actions on the result string. Either by suppressing characters or adding characters or by positioning within the pattern result string.

8.2 $MATCH command

syntax :

```
$MATCH statement = dollarmat
          ( string | identifier | '<' concatenation '>' )
                   ( pattern [ '=' concatenation ] )
                    [ ':' expression ]
```

The identifier may be :

- string variable.
- string array element.
- read/write inline string function.
- NODAL string defined function.

The purpose of the command is to match the pattern against the string or concatenation which precedes the pattern. When the statement includes a pattern assignment then this pattern assignment is only executed when the match succeeds.

A match will succeed when any of the possible result strings defined by the pattern is completely present in the object string defined immediately after the $MATCH command. The found result string may be preceded or followed by any number of characters.

The following examples will clarify these concepts.

```
$MATCH "OYES" "YES"
```

will first try to match YES with the first three characters in the match string OYE. When this has failed it will move its position to the second character and find YES which matches. The second string can be replaced by a pattern.

```
$PAT P1="YES"; $MATCH "OYES" P1
```

which again would have matched with YES. It is possible to reassign another string to the object string, if the latter is an identifier.

```
$SET ST="OYES"; $PAT P1="YES"; $MATCH ST P1="NO"
```

After the first match the contents of the matched substring is now replaced with NO and ST will contain ONO.

In case of mismatch the line following the $MATCH command will be executed. When the $MATCH statement contains a ':' then a GOTO the line specified by the expression will be executed.

```
10.1 $PAT P1="YES"; $SET ST="OYES"
10.2 WHILE 0=0 ;$MATCH ST P1="NO":10.4; NODAL statement
10.4 END
```

In the above example the first time a match will occur, and the NODAL statement will be executed; but the second time no match will occur and a GOTO line 10.4 will be executed, which ends the program.

Within the pattern, variables can be specified into which the result of a match will be stored. Two types of assignments can be discerned :

- immediate assignment, preceded by a '$'
- deferred conditional assignment, preceded by a '.'

In the first case after every successful partial match, the result is stored in the item specified after the $ sign. In the second case the assignment only takes place when the whole pattern has been matched. Finally it is possible to visualise the results of the matches, as is clarified in the following example :

```
$PAT P1 = ("YES" $S1 "NO") .S2
$MATCH "YES, YES, YESNO" P1
```

the string YESNO will be matched against the one specified in the $MATCH command. Three times YES will be matched and immediately stored in S1 and YESNO will once be stored in S2. When the command had been :

```
$MATCH "YES, YES, YES NO" P1
```

then S1 would still have received YES three times but S2 will never be filled as YESNO is not specified. When the pattern P1 is modified to :

```
$PAT P1 = ("YES" .S1 "NO") .S2
```

then in the first case S1 will receive YES only once and S2 will recieve YESNO. The end result is not different. However in the second case both S1 and S2 will receive nothing.

A final example is :

```
$PAT P1 = "BE" ! "BEA" ! "BEAR"
$PAT P2 = "RO" ! "ROO" ! "ROOS"
$PAT P3 = "DS" ! "D"
$PAT P4 = "TS" ! "T"
$PAT FI = P1 P3 ! P2 P4
```

The result strings of this pattern are :

```
BE        DS              BEDS   BEADS BEARDS
BEA       D               BED    BEAD  BEARD
BEAR
------------------- ==>
RO        TS              ROTS   ROOTS ROOSTS
ROO       T               ROT    ROOT  ROOST
ROOS
```

yielding a total of 12 possible result strings.

8.3 Pattern functions

A subdivision of three types of pattern functions can be made :

- position dependent functions (POS, RPOS, TAB, RTAB, LEN)
- character dependent functions (SPAN, BREAK, ARB, ANY, NOTANY)
- control functions (ABORT, FAIL)

The POS and RPOS functions serve to position patterns at a certain place in the object string. POS(0) will force a pattern at the beginning of the object string and RPOS(0) will position the pattern at the end of the object string.

The pattern :

```
$PAT P1 = "CAR" ! "CARD" ! "CARDS"
```

will ascertain that one of the three strings is present in the object string. However the pattern :

```
$PAT P2 = POS(0) P1
```

ascertains that one of these strings is present at the beginning of the string. Actually the two alternatives have no real meaning here.

The function LEN assures that a string has a certain length. For example the pattern :

```
$PAT P1 = '(' LEN(5) ')'
```

assures that the object string contains somewhere the string

```
(xxxxx)
```

where x stands for any character.

The functions TAB and RTAB match all characters in the object string until the position defined by one of the two functions. TAB matches all characters up to position i, while RTAB matches all characters from position i until the end of the string. The pattern

```
TAB(0) RTAB(0)
```

always matches the whole of a string.

The functions are specially useful in breaking out pieces of strings. For example

```
TAB(4) .A1
```

will assign the first four characters of the object string to A1. As such it is convenient to obtain separate parts of object strings of which the format is known beforehand. It should be remembered that it is not allowed to move the cursor backward during pattern matching. The pattern :

```
LEN(5) TAB(4)
```

will fail, because TAB(4) positions the cursor to a position which is already passed.

SPAN and BREAK are pattern functions which match runs of characters. Patterns which describe a run of characters, or a run of blanks, or a run of numbers can be described by using SPAN :

SPAN(NUM) will match a run of numbers. SPAN(ALPHA) will match run of capital letters. SPAN(' ') will match a run of blanks.

Patterns to describe anything up till a character are constructed with BREAK.

BREAK(ALPHA) will match anything until a capital letter is met. BREAK(NUM) will match anything until a decimal digit is met. BREAK(',.;:!?') will match anything up till a punctuation mark.

SPAN may be thought of as the streaming of the cursor as long as characters contained in the argument are met; BREAK is the streaming of the cursor until characters present in the argument are met. These functions are very useful for the extraction of strings which are separated by special characters or which contain a set of known characters.

For example the pattern :

```
$PAT P1 = BREAK(' ') .A1 SPAN(' ') BREAK('.') .A2
```

will have the following action on

```
$MATCH "IT   RUNS." P1
```

A1 will contain IT, SPAN will remove all blanks, then A2 will contain RUNS, while the end of the string should contain a dot. To ascertain that this is the end of the string a RPOS(0) might have been added.

The ARB function will match any character up till the character followed by it. It constitutes an AND of SPAN and BREAK. For example

```
$PAT P1 = "A"  ARB  "D"
```

will match all strings containing an A followed by a D somewhere.

The ANY and NOTANY functions are pattern functions which match single characters. ANY matches any character appearing in its argument. NOTANY matches any character not appearing in its argument. Actually

```
ANY(' #$Z&')
```

is equivalent to :

```
' ' ! '#' ! '$' ! 'Z' ! '&'
```

The order of the characters in the argument is irrevelant to the result of the match. For example :

```
NOTANY('STRUCTURE')
```

is equivalent to :

```
NOTANY('CERSTU')
```

FAIL is a pattern function which will force any pattern to fail a match. FAIL causes all alternatives to be tried. This function is generally used if the programmer wants the interpreter to try a number of alternatives and visualise them. A good example is

```
$PAT P1 = 'IS' ! 'SI' ! 'IP' ! 'PI'
$MAT "MISSISSIPPI" P1 $ OUTPUT FAIL
```

will result in the printing of

```
IS SI IS SI IP PI
```

While the command

```
$MAT "MISSISSIPPI" P1 $OUTPUT
```

would have resulted in the printing of IS only.

The ABORT function forces a match to fail entirely. No alternatives are tried. This is useful for conditional pattern matching constructs. When only matches of a string with a length less than 12 is required the following pattern would have the desired effect :

```
$PAT P1 = LEN(12) ABORT ! P2
```

In this case the match fails when the object string has more than 11 characters. When the string has less than 12 characters the LEN functions fails and the alternative P2 is tried.

CHAPTER 9

Interactive command I/O streams

9 Interactive command I/O streams

NODAL maintains two main input and output device streams. For the interactive version these streams are automatically set to the current input and output device : the terminal on which NODAL was activated. After every execution of an interactive NODAL line or after the termination of a NODAL program (return to interactive mode) the output device stream is reset to this terminal. The input device will be reset to terminal when an end of file was met on the current input stream.

Two functions are associated with these streams :

- IDEV which denotes the input device stream.
- ODEV which denotes the output device stream.

For example a listing can be made with the command :

```
SET ODEV=OPEN("W","FILE"); LIST
```

In this interactive command the file is opened and the output device stream is set to this file. The LIST command then makes a listing of the program, which is sent to the file by the NODAL interpreter. When the line which contains the ODEV set command is finished (in the example the LIST command), the file is closed and the interactive I/O stream is reset to the terminal.

With the line :

```
SET IDEV=OPEN("R","FILE")
```

the file is read by the command interpreter until an end of file is met. Then the file is closed and IDEV is reset to the terminal. The individual commands which work with the terminal I/O will be described below.

9.1 TYPE command

syntax :

TYPE statement = typecom concatenation { ',' concatenation }

The concatenations separated by commas are evaluated and the result strings are concatenated to an intermediate string, which is then sent to the current I/O device.

Example :

```
T 1
```

will result in

```
     1
```

and

```
      T 1,2 "HELP" ! , 4.5 \10 \13, " WELL DONE"
```

will result in

```
     1      2    HELP
     4.5
 WELL DONE
```

The rules are the same as laid down in chapter 6.2 about concatenations.

The maximum length of the printable result string is 80 characters.

9.2 ASK command

syntax :

```
ASK statement = askcom [ string ] identifier
                    { [ string ] identifier }
```

The identifier should be :

- simple variable.
- integer or real array element.
- read/write in-line functions.
- write only in-line functions.
- NODAL defined functions.

The purpose is :

- Output string to current output device and add ':'.
- Get from the current input device an expression.
- Evaluate the expression and store it in the identifier.
- When identifier does not exist create a simple variable.

Example :

```
    ASK "FIRST VALUE" A "SECOND VALUE" B
```

will result in the printing of

```
FIRST VALUE :
```

It is then possible to type in the expression, either a simple value (e.g. 1) or a more complex expression like

```
        (2 * B + SIN(PIE/4))
```

After the carriage return the expression is evaluated and the result is stored in A. The interpreter will now display the second message and wait for an expression to be typed in.

The addition of strings is optional. The same thing could have been written as :

```
ASK A B
```

Then only the colon would have been printed to prompt for the typing of an expression.

Several expressions for the same ASK command can be answered immediately by typing them already the first time, separated by commas.

```
ASK "FIRST VALUE" A "SECOND VALUE" B " THIRD VALUE" C
```

can be answered immediately after the prompt:

```
FIRST VALUE : 1, 2, 3
```

The values 1,2 and 3 will be assigned to A, B and C respectively.

By answering with an asterix (*) the value of the parameter will stay unchanged; no assignment will take place. The typing of CR will yield the same result as the typing of an asterix.

9.3 $ASK command

syntax :

```
$ASK statement = dollarask [ string ] identifier
                         { [ string ] identifier }
```

identifier can be :

- simple string variable
- string array element
- read/write string function
- NODAL string defined function

The purpose of this command is

- Output string and ':' to current output device.
- Edit a string on the input device.
- When element exists edit old string.
- Store this string into the identifier.
- When the identifier does not exist create a simple string variable.

It is possible to specify several string identifiers and prompt texts within one $ASK command.

To obtain the old string the <CNTRL> P should be typed twice.

Example :

```
$ASK "ONCE" S1 "TWICE" S2 "THREE TIMES" S3
```

will result in :

```
ONCE : <edit string>
TWICE : <edit string>
THREE TIMES : <edit string>
```

Due to the editing of the old strings, it is impossible to specify all three strings in one go as in the ASK command.

The texts for the prompts are optional and can be left out. The following are also valid :

```
$ASK S1 S2 S3
$ASK S1 S2 "THREE TIMES" S3
etc .....
```

9.4 LIST command

syntax :

```
LIST statement = listcom {identifier|expression|listid}
 listid = 'ALLP' | 'ALLV' | 'ALL' | 'ALLD' | 'ALLR'
```

Identifier can be :

- array name.
- variable name.
- NODAL defined function name.

Expression may denote :

- group of lines
- one line

The LIST or LIST ALLP command will list all lines contained in the NODAL program, which is currently present in the work area. Lines which have been executed are stored in a preinterpreted form. These lines are preceded by the ∂ sign.

LIST ALLV will list all variables present in the work area of the program. First their name then their type is displayed. For simple variables this is followed by the value of the variable, while for other types of variables the parameters are displayed or the dimension of the arrays.

LIST n with 1 <= n <= 99, will list all lines belonging to group n of the NODAL program currently in the work area.

LIST n.m with 1<=n<=99 and 1<=m<=99 will list the line n.m which is currently in the work area.

When group n or line n.m is not present the list will be empty.

LIST ALLD will list the names of all NODAL defined functions which are currently defined.

LIST ALLR will list all in-line NODAL functions which are present and at the disposal of the NODAL defined functions and programs.

LIST NAME will display the information about NAME dependent on its type.

- When NAME is a simple variable, its type and contents will be displayed.
- When NAME is an ARRAY, its type and dimension will be displayed.
- When NAME is a NODAL defined function, its type and parameters will be displayed.

CHAPTER 10

In-line functions and NODAL defined functions

10 In-line functions and NODAL defined functions

Two types of functions exist :

- NODAL defined functions which are constituted from NODAL lines.
- in-line functions which are compiled additions to NODAL.

The first one is created interactively and can be edited with the editor on an individual basis. Their names are shown with the LI ALLD command.

The second one is created from MODULA-2 and FORTRAN. They adhere to a strict parameter scheme which emulates the NODAL parameter scheme to be independent of the compiler implementations which supports them. Their generation and loading into the interpreter is dependent on the host system and are described in the individual function manuals.

10.1 Function types and parameter types

Several types of in-line functions exist :

- Read/write functions :
Functions may be used in an expression and will return a numeric value, or functions can be the destination of an assignment statement.

- Write only functions :
Function may not be used in an expression. Function is only used as destination of an assignment.

- Read-only functions:
Function is only used in an expression,and returns a numeric value.

- Call functions :
Function is used in CALL statement. All information from/to the function is passed in the parameters. The functions cannot be used in expressions or assignments.

- Pattern functions :
Functions which are used in pattern definitions (see pattern chapter).

- Read/write string functions :
Functions may be used in a concatenation and will return a string value, or may be used as the destination of a string assignment.

-Write-only string functions :

Function is only used as the destination of a string assignment.

-Read-only string functions :
Function is only used in a concatenation and returns a string value.

There are three types of NODAL defined functions :

- Read/write NODAL defined functions :
Functions may be used in an expression and will return a numeric value, or functions can be the destination of an assignment statement.

- Read/write NODAL string defined functions :
Functions may be used in a concatenation and will return a string value, or may be used as the destination of a string assignment.

-Call NODAL defined functions :
Function is used in CALL command. All information from/to the function are passed through the parameters. The functions cannot be used in expressions or assignments.

Each function can have up till 8 parameters. The parameter types for in-line functions are :

- Real value : A REAL value is passed to the function. The value is copied to a space local to the invoked function. The argument may be any expression.

- Integer value : A 16 bit integer is passed to the function. The argument may be any expression which is converted to a 16 bit integer and stored in a space local to the invoked function. When the value of the expression is too large the value is truncated to the highest possible value.

- String value :
A string with its NODAL string descriptor is passed to the function. The argument may be any valid concatenation with a maximum of 80 characters. The concatenation result (a string) is copied to a space local to the invoked function.

- NODAL reference :
A pointer to the data descriptor of the referred NODAL element. The argument must be an existing NODAL element.

- Real reference :
A pointer to a REAL value. The argument can be a REAL variable or a REAL array element. The invoked function can change the value of the argument.

-Integer reference :
A pointer to an integer value. The argument can be a integer array element. The invoked function can change the value of the argument.

- Real array :
A pointer to the start of a REAL array. The argument can be a REAL array.

- Integer array :
A pointer to the start of an integer array. The argument can be an integer array.

- NODAL name :
A 6 byte name. The 6 characters are copied to a space local to the invoked function.

Three types of parameters exist for the NODAL defined functions :

- Value parameter :
A local variable of the specified name is created in the NODAL defined function. Its contents is the result of any expression specified in the argument. Its contents may be changed in the called NODAL defined function, but the new value is not transmitted to the calling routine.

- Reference parameter :
A descriptor of the specified name is created local to the NODAL defined function. The contents of this variable and its type is determined by the argument. Arguments may be : Real variable, Real array or Integer array. Its contents is directly changeable in the calling routine.

- String parameter :
A local variable of the specified name is created with the contents determined by the string which is used as parameter. The string may be modified by the called function, but the new value is not transmitted to the calling routine.

Default parameters are allowed in NODAL defined functions and NODAL in-line string functions. In the NODAL defined function they are called PAR1 and PAR2; In the in-line functions their name is determined by the function coding.

Examples for parameters in NODAL defined functions will be shown in chapter 10.3, while examples for in-line functions can be found in the function manuals.

10.2 CALL command

syntax :

```
CALL statement = calcom identifier [ '(' paramlist ')']
paramlist = param {, param}
```

param depends on the type of parameter.It may contain expressions, concatenations or variable references.

The maximum number of parameters is eight.

Functions can be invoked by the CALL command. The addition of CALL is optional. The name of the function is sufficient to execute the function.

For example, both

```
>CALL FUNC or
>FUNC
```

are allowed to invoke the parameterless function FUNC.

Bothin-line call functions and call NODAL defined functions can be invoked with this command.

When NODAL executes a function it will first evaluate all parameters and put them on a local stack. Afterwards it will execute the function. After execution control returns to the NODAL interpreter which will execute the statements following the CALL command. When an error is detected by the function, a NODAL error return is executed.

10.3 DEFINE command

syntax :

```
DEFINE statement = defcom '-' ( call | fun | str )
                                identifier [ '(' paramdefs ')' ]
 call = 'C'|'CA'|'CAL'|'CALL'
 fun  = 'F'|'FU'|'FUN'|'FUNC'
 str  = 'S'|'ST'|'STR'|'STRI'|'STRIN'|'STRING'
 paramdefs = funparam { ',' funparam }
 funparam = ( 'V' | 'R' | 'S' ) '-' identifier
```

The maximum number of funparams is eight.

The identifier in the funparams denotes the name of the parameter by which it is referred inside the NODAL defined function. Even for reference parameters the name of the parameter inside the NODAL defined function is the name defined in the DEFINE command, while the name of the actual parameter may be different.

The identifier of the name of the NODAL defined function may be any name of six characters which is not already defined.

The DEFINE command will create a NODAL defined function header according to the command specifications. The contents of the current NODAL program will be copied into this NODAL defined function. After creation the function can be edited with the NODAL editor. Three types of functions can be created :

- Call functions, with DEF-C.
- Read/write functions, with DEF-F.
- String read/write functions, with DEF-S.

Some examples will follow below.

The command :

```
DEF-C FUNC(R-A)
```

will define a NODAL defined function FUNC with one reference parameter called A. It should be noted that the type of the variable is not specified.

The coding of the function can be created by calling the editor with:

```
EDIT FUNC
```

The NODAL lines can be supposed as:

```
1.1 FOR I=1,10;SET A(I)=I
```

The calling program may call this function in the following way :

```
DIM-I BB(20); FUNC(BB) or
DIM BC(20); FUNC(BC)
```

In both cases the first 10 items of the arrays are filled by FUNC.

However in the following two cases errors will be reported during the execution of FUNC :

```
SET B=20; FUNC(B)

DIM B(5); FUNC(B)
```

The following statement will also cause an error at the invoking of FUNC

```
FUNC(20)
```

Here no reference variable is mentioned so a parameter error is detected. The following statement is also not allowed.

```
DIM B(20); FUNC(B(10))
```

References to single array items are not allowed.

The following example shows the use of V-variables :

```
DEF-C FUNC(V-A)

1.1 SET A=A*2; TYPE A
```

This function can be called with an expression as parameter.

```
FUNC(2*10)
```

will result in the typing of 40.

```
SE A=4; FUNC(A)
```

will result in the typing of 8, but the value of A remains unchanged as the A inside FUNC is a local copy of the expression specified as parameter.

With a string as parameter the following example will show some of the constraints inherent to its use :

```
DEF-C FUNC(S-S)

1.1 TYPE 'HELP ' S
```

the call FUNC('ME') will result in the typing of HELP ME.

Inside FUNC it is allowed to change the contents of S, but its value will not be returned to the invoking code. For example :

```
1.1 $SET S= "HELP" S; TYPE S
```

will result in the modification of S inside the function but not in the program or function which called FUNC.

Apart from the specified parameters other variables are automatically generated inside a NODAL defined function when it is invoked by the NODAL interpreter.

PAR1, PAR2 are created, when no parameters were specified in in the DEFINE command. They are the default parameters and need not be specified in the NODAL call. When not specified in the call their value is zero. For example :

```
DEF-F SUM

1.1 VALUE PAR1+PAR2
```

will yield the following results :

```
SUM           will yield 0
SUM( 6)       will yield 6
SUM( 22, 4*3) will yield 34
```

When no default parameters are wanted the function should be specified with

```
DEF-F FUNC()
```

The specification of parameters in the call will then provoke an error.

VALUE is a local variable, which is created when the function is invoked as the destination of an assignment. VALUE will contain the result of the expression or of the concatenation which is assigned to the function.In functions created with the DEF-F command it will be a read-only variable and for functions created with DEF-S it will be a string variable.

FLAG is always generated for functions created with DEF-S and DEF-F to indicate if a value has to be returned from the function or has to be assigned to the function. FLAG is 1 if a value must be returned. FLAG is -1 when a value has to be assigned. In the latter case the value can be found in the variable VALUE.

10.4 VALUE command

syntax :

VALUE statement = valcom expression

This command is associated with with NODAL defined functions defined by the DEF-F command. The command permits to return values to a NODAL expression, when the invoked function is part of this expression. The VALUE command also terminates the execution of this function and acts as an END. Example the function TWO

```
DEF-F TWO()

1.1 VALUE 2
```

Then the command TYPE 20*TWO will result in the typing of 40.

More complex expressions are also allowed in the VALUE statement.

```
VALUE 20*sin(PIE)/A
```

is a perfectly valid statement when A is defined.

In the functions a test can be done to determine if the function is invoked in read or in write mode. A more elaborate coding for the function TWO is then :

```
1.1 IF FLAG = 1; VALUE 2
1.2 TYPE VALUE*2
```

In this case first the variable FLAG is tested on the way the function is invoked. If the function is invoked in read mode then the value two is returned and control returns to the invoking NODAL line. If the function is invoked in write mode then the VALUE variable is multiplied by two and typed.

The NODAL defined functions can be called recursively with return of a numeric value by the VALUE command. The standard factorial example will be used :

```
DEF-F FAC(V-A)

1.1 IF A > 1; VALUE FAC(A-1)*A
1.2 VALUE 1
```

The command T FAC(5) will yield the number 120, which is correct. The same non recursive result can be coded as :

```
1.1 SE R=1
1.2 FOR I=1,A; SE R=R*I
1.3 VALUE R
```

10.5 $VALUE command

syntax :

$VALUE statement = dollarval concatenation

This command serves to return strings when the function is used in a concatenation. The command is associated with functions defined with the DEF-S command. Example the function TWICE :

```
DEF-S TWICE(S-ST)

1.1 $VALUE ST " " ST
```

The command TYPE TWICE("HELP") will result in the typing of HELP HELP.

The $VALUE command also acts as an END command inside the invoked function. $VALUE returns the concatenation result to the calling program and returns control to the invoking NODAL line.

The TWICE function can be extended with a test to determine if it is called as an assignment destination or as part of a concatenation.

```
1.1 IF FLAG = 1; $VALUE ST " " ST
1.2 TYPE VALUE " " VALUE
```

In this case the response to TYPE TWICE("HELP") will be the same as for the function defined above, but the command

```
$SET TWICE(" ")= "HELP"
```

will additionally result in the typing of HELP HELP.

10.6 OPEN command

syntax :

OPEN statement = opencom identifier

The identifier must be the name of a NODAL defined function.

This command moves the NODAL defined function to the work area. First the current program is erased and then the contents of the NODAL defined function are copied into the work area.

Example :

```
OPEN FUNC
```

will do an effective ERASE ALL, get the contents of FUNC, copy it into the work area, delete the function FUNC and remove its name from the function list.

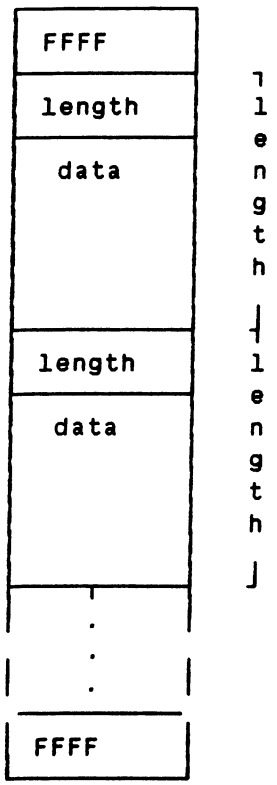
CHAPTER 11

File handling

11 File handling

A common storage scheme is used to store NODAL programs and data on files. A set of commands is available to store NODAL lines and data on files and retrieve them afterwards.

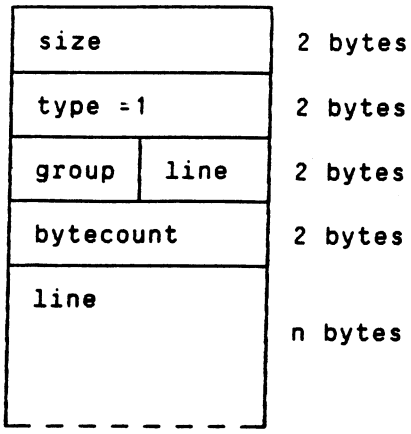
The general structure of the file is organized in 16 bit items. It starts with a 16 bit integer which contains FFFF and ends with a 16 bit integer containing FFFF. In between all NODAL data are stored, each item is preceded by its length (16-bits) in 16 bit words. See the figure below :



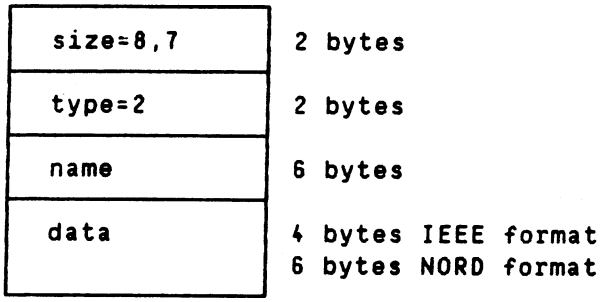
The data parts can contain the following items :
- Variables or arrays.
- NODAL defined function header.
- NODAL lines.
- Patterns.

Their individual structure will be shown below.

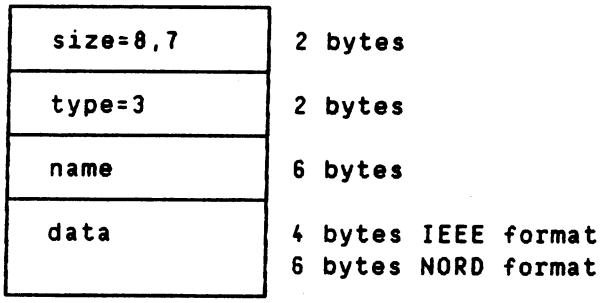
NODAL line



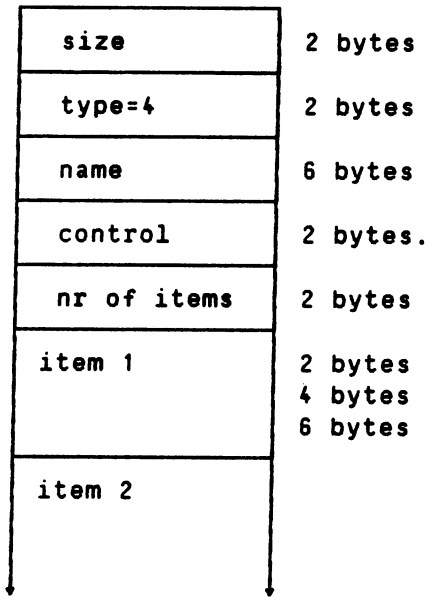
variable

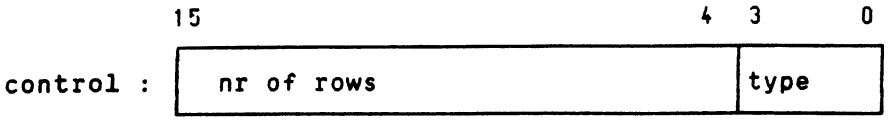


RO variable



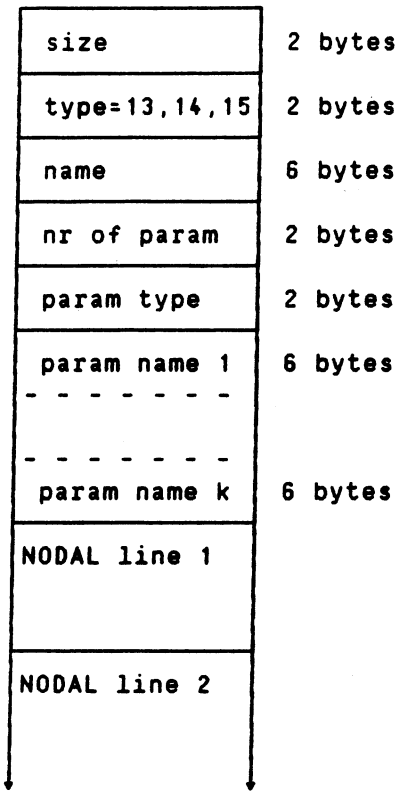
Array





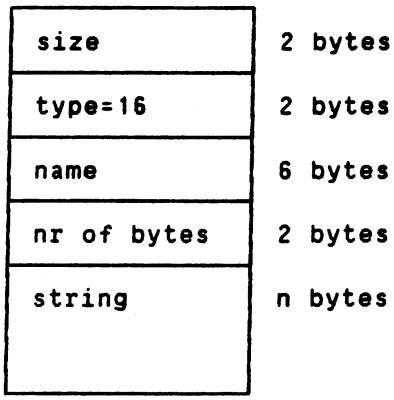
type = 1 : integer 16 bits
2 : IEEE 32 bit
3 : NORD format 48 bits
6 : integer 32 bits

NODAL defined function



nr of param = FFFF : default parameters
param type = 8 fields of 2 bits 1 : value param
2 : ref param
3 : string param

String variable



String array

	size	2 bytes
	type=17	2 bytes
	name	6 bytes
	no of items	2 bytes
item 1	array index	2 bytes
	nr of bytes	2 bytes
	string	n bytes
item 2	array index	2 bytes
	nr of bytes	2 bytes
	string	n bytes
item 3		

Pattern variable

size	2 bytes
type=19	2 bytes
name	6 bytes
nr of bytes	2 bytes
string	n bytes

11.1 SAVE command

syntax :

```
SAVE statement =
  savecom  filename  {line | group | identifier | saveid}
  saveid = 'ALLP' | 'ALLV' | 'ALLD' | 'ALL'
```

Identifier may be :

- variable.
- array.
- NODAL defined function.
- pattern variable.

The syntax of the filename is completely determined by the host operating system.

The actions of the the SAVE command are :

- OPEN the specified file for write
- Write a minus 1 (hex : FFFF) to file
- write all specified NODAL elements to file
- Write a minus 1 (hex : FFFF) to file
- Close file

When nothing is specified behind the filename the default ALLP is assumed.In this case all lines belonging to the current NODAL program are stored on the specified file.

SAVE filename ALLD means: store all NODAL defined functions onto specified file.

SAVE filename ALLV means : store all NODAL variables onto specified file.

SAVE filename ALL is equivalent to SAVE filename ALLP ALLV.

SAVE filename NAME will store the specified item on the file.

A number may equally denote a line number or a group.

Several items may be specified in one SAVE command. The maximum number is determined by the number of characters in one NODAL line (80). When two SAVE commands to the same file are executed, the information of the first command is overwritten by the information of the second command.

An example is:

```
SAVE file ALLP AR 2
```

will save the contents of the current NODAL program the array AR and the group 2 onto the specified file. Consequently the group 2 will be stored two times on the file (ALLP and 2).

11.2 LOAD command

syntax :

LOAD statement = loadcom filename

The syntax of filename is completely determined by the host operating system.

The actions performed by this command are :

- Open the file for read.
- Read the first FFFF.
- Read an element from the file.
- If the element is present in the program area, erase it.
- Store the element in the program area.
- Continue reading/storing of elements until FFFF is read.
- Close the file.

The loading of elements is done on an element by element basis, especially the lines are loaded on a line by line basis. When a whole group was saved by the former SAVE command, the lines are still replaced on a line by line basis.

If the LOAD command is executed from a NODAL program, care should be taken that the lines which are currently active, are not replaced by lines contained in the file. When this happens, the interpreter will crash in an unpredictable way.

The LOAD command cannot be invoked from a NODAL defined function.

11.3 OLD command

syntax :

OLD statement = oldcom filename

The syntax of the filename is completely determined by the host operating system.

The actions performed by this command are :

- Open the file for read.
- Clear the current program/work area.
- Read the first FFFF.
- Read an element from the file.
- If the element is present in the program area, erase it.
- Store the element in the program area.
- Continue reading/storing of elements until FFFF is read.
- Close the file.

It is not allowed to use the OLD command in program mode. I.E. it is not allowed to use the OLD command either in a NODAL program or in a NODAL defined function

11.4 RUN command

syntax:

RUN statement = runcom ['[' line ']'] [filename]

The syntax of filename is completely determined by the host operating machine.

Together with the DO command, the RUN command serves to pass from interactive mode to program mode. It is possible to specify a line number at which program execution should start. The line number is enclosed within square brackets []. The default number is the lowest NODAL line actually present.

The command allows to first load a file from the specified filename after which execution starts at the specified or default line number. When the specified line number is not present, execution will start at the line with the next highest line number. Before the file is loaded the current NODAL program is erased, which is equivalent to a OLD command. So the comand:

```
RUN [line] file
```

is equivalent to:

```
OLD file; RUN [line]
```

and not equivalent to:

```
LOAD file ; RUN [ line]
```

It should be noted that the RUN command erases the present context of the running program. This means that all DO and FOR/WHILE contexts are lost. Consequently great care should be excercised in using the RUN command in program mode. However the command RUN [n.m] can be used freely throughout the NODAL program.

For the case of a RUN command with a file name specification the current program is erased and consequently also the line which invoked the RUN command. Here the same problem appears as with the OLD command. There is only one instance which allows the use of the RUN command with a file name specification in a program :

```
N.n $DO 'ERASE ALL; RU [M.m]File'
```

The RUN command cannot be invoked from a NODAL defined function.

11.5 OVERLA command

syntax:

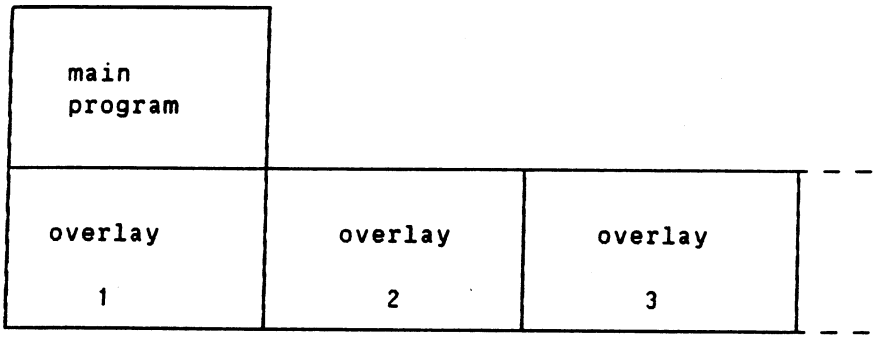
```
OVERLA statement = overcom filename [ paramspec]
paramspec = expression {(' '|',') expression }
```

Up till 8 expressions can be specified.

The syntax of filename is completely determined by the host operating machine.

The action of the OVERLA statement is the following :

The present NODAL program context is saved. The program contained within the specified file is loaded and executed without interference with the invoking NODAL program. After execution of the program specified in the OVERLA command, control returns to the statement following the OVERLA statement. The program, data and context of invoked overlay program has been erased. The figure below shows the actions in a schematized way



The programs specified in overlay 1, 2, .. use each other's work area and can only be executed one after the other. They are not aware of the existence of the main program and neither is the main program aware of them.

The OVERLA command provides a useful way to save space and to execute a certain number of programs in a fixed specified order.

To pass parameters from the main program to the overlays and vice-versa the REAL array ARG(1) .. ARG(16) and the string variable STRARG are at the disposal of the programmer. These 16 numeric values and the string variable always keep the same last assigned value independent of the program and overlay context. They are super global variables which are always visible and accessible (also to NODAL defined functions).

To facilitate the use of the ARG's in the context of the OVERLA command an automatic assignment is provided for.

The first expression following the file name specification is assigned to ARG(1), the second is assigned to ARG(2), etc up till the eighth expression.

For example the statement :

```
OVERLA myfile 12/3, 5, SIN(0)
```

will assign the value 4 to ARG(1), the value 5 to ARG(2) and the value 0 to ARG(3); then it will invoke the program contained in myfile and execute it as a single NODAL program.

The OVERLA command can only be invoked from a NODAL program.

CHAPTER 12

Network access

12 Network access

NODAL was originally designed for the SPS control network. To exploit the network facilities, some network oriented commands are part of the NODAL language. They are mainly destined to invoke the interpreter on some remote computers, execute NODAL code on this computer and return eventual results.

12.1 IMEX command

syntax :

IMEX statement = imexcom '(' comp ')' NODAL statements

comp can be the remote computer name or the computer number. Its actual specification depends on the network implementation.

NODAL statements are the usual NODAL statements contained within one line and separated by semicolons.

The IMEX command will send the whole line following the computer specification to the specified computer. The destination computer will invoke the local NODAL interpreter which will execute the NODAL statements contained in the remainder of the line. The input stream of the NODAL lines in the remote computer is undefined, while the output stream is set to the output stream of the invoking source NODAL interpreter. The example below:

```
IMEX(COMP)TYPE 1
```

will result in the sending of the command TYPE 1 to the specified computer. The result of the execution: " 1" will be typed on the invoking interpreter output device. The interactive result of the command will then be :

```
>IMEX (COMP) TYPE 1
     1
>
```

And an example with two statements is:

```
> IMEX(COMP) FOR I=1,4; TYPE I !
     1
     2
     3
     4
>
```

the IMEX command is useful to inspect visually the return of functions which are unavailable on the local computer. (Especially in the context of equipment functions).

12.2 EXECUT command

syntax:

EXECUT statement = execcom '(' comp ')' (line | group)
{ identifier | line | group }

comp can be the remote computer name or the computer number, its actual specification depends on the network implementation.

Identifier may be:

- Variable.
- Array.
- NODAL defined function.

The lines, groups, variables, arrays or NODAL defined functions specified in the EXECUT command are sent over to the specified computer for execution. The destination computer interpreter will receive, load and execute these NODAL elements. It is compulsory to send at least one NODAL line or NODAL group to the destination computer to specify the start of execution.

No input or output streams are associated with the remote program. The TYPE or ASK command will have no net result, unless ODEV or IDEV streams, local to the target computer, are specified. Data are retransmitted to the invoking program with the REMIT command.

An example will be shown below:

```
EXEC (COMP) 20.1 FUNC A

20.1 FUNC(A)
```

In this example a NODAL defined function FUNC is present in the invoking program. This function together with the variable A (which should be present) and the line 20.1 are sent to COMP. In COMP the line 20.1, the function FUNC and the variable A are loaded. Execution starts at line 20.1, which invokes FUNC with as parameter A. After execution of FUNC no more NODAL program is present and the remote program stops.

12.3 REMIT command

syntax:

REMIT statement = remitcom identifier { identifier }

Identifier may be:

- Variable.
- Array.

The REMIT command is only used in connection with the EXECUT command. Its main purpose is to send back NODAL variables to the invoking interpreter program, and then finish the remote program execution. The REMIT contains an implied QUIT command.

The example below will show the link between EXECUT and REMIT commands

```
EXECUT (COMP) 10

10.1 DIM A(10)
10.2 FOR I=1,10;SET A(I) = I
10.3 REMIT A
10.4 SET B=5
```

The EXECUT command will send group 10 over for remote execution. In the destination computer the array A is defined and is filled with its index values. The array A is then sent back to the invoking computer and the execution of the remote program is stopped (i.e. line 10.4 is never executed).

12.4 WAIT command

syntax:

WAIT statement =
waitcom ('(' comp ')' | '-T' expression | '-C' expression)

comp can be the remote computer name or the computer number; its actual specification depends on the network implementation.

Dependent on the syntax of the command the actions undertaken are different.

The WAIT computer command is used in connection with the EXECUT command in the sending computer. It actuallly completes the sequence of:

- Send program, data. (EXECUT local)
- Execute program, return data. (REMIT remote)
- Wait for data. (WAIT local)

The WAIT command stops the execution of the program and synchronises with the reception of the data specified by the computer name. When no EXECUT was done to this computer an error is generated. Together with the EXECUT and WAIT command a series of simultaneous EXECUTs is now possible. A program can start a series of remote EXECUT commands, do something else and then process the returned data. It should be noted that data are returned by name and that the same name can be sent back from different computers. The distinction between the data is then determined by the WAIT command.

At the end of an interactive NODAL command containing an EXECUT, or at the END of a program doing EXECUTs, implicit WAITs are done by the NODAL interpreter to receive all data from all currently active EXECUT commands.

A final example will show how the three commands EXECUT, REMIT and WAIT can be used in the context of a beam scanner.

```
1.1 ASK 'INITIAL POSITION' IP "FINAL POSITION" FP
1.2 SET P=IP; EXECUT (COMP) 3.2 P
1.3 FOR P = IP+1,FP; DO 2

2.1 EXECUT (COMP) 3 P; WAIT (COMP)
2.2 TYPE "POSITION=" P-1 "CHARGE=" A

3.1 SET A=MINSCN(2)
3.2 SET MINSCN(2,#PSN)=P
3.3 REMIT A
```

Two other types of WAIT are also possible :

WAIT-T expression: will wait for the number of seconds specified in the expression.

WAIT-C expression: will wait for the machine event specified by the expression.

CHAPTER 13

Utilities

13 Utilities

A certain number of utilities is available for debugging and information about the NODAL interpreter.

13.1 HELP command

syntax :

HELP statement = helpcom

The typing of HELP will result in a listing of the possible commands and their abbrevations.

13.2 ?ON and ?OFF commands

syntax :

?ON statement = oncom
?OFF statement = offcom

The ?ON and ?OFF commands serve to activate or to suppress the post mortem debug display information.

After the typing of ?ON a the post mortem dump will be displayed when:

- a NODAL error is detected.
- a breakpoint is met.

The ?OFF command will suppress these displays.

The default is ?OFF when NODAL starts up.

13.2.1 Post mortem dump information

The post mortem dump consists of five windows or chapters. Each window displays information about the actual status of the program. There is one main window. From the main window it is possible to go to one of the four secondary windows. From a secondary window one can go to the main window or any of the secondary windows. The contents of each window is:

- Main window displays:
  * The halt cause (error, breakpoint)
  * Halt ocurred during execution or syntax analysis.
  * The line where the halt ocurred.
  * The function or main program in which halt ocurred.

- Data window displays:
  * The data at a given level of execution.

- Text window displays:
  * 20 lines of text with erroneous or last active line at top.

- Calls window displays:
  * The order and level of currently active functions.

- Flow window displays:
  *The order in which lines are activated at a given level.

Example:

A main program is activated:

```
1.10 SE A=[[FF
1.20 DO 2
1.30 END

2.10 F1(A)
```

which calls a function F1 at line 2.10:

```
F1(V-X)

1.10 SE A1=X
1.20 FOR I-0,5;DO 2
1.30 END

2.10 SE C1=A1/(2-I)
2.20 TYPE C1 !
```

At line 2.10 an error will be provoked :

The main window will display the error code (division by zero) at line 2.10 in function F1

The data window will display the data of level 1 in F1 and the data at level 0 in the main program

The text window will display lines 2.1 and 2.2 of F1, and line 2.1 for level 0.

The calls window will show that the main program called F1.

The flow window will show that line 1.2 called line 2.1 in F1 at level 1, and that line 1.2 called line 2.1 in main at level 0.

At each window the following commands are available:

In Main window:

* C to go to Calls window.
* D to go to Data window.
* T to go to Text window.
* F to go to Flow window.
* G to continue execution after breakpoint.
* H to display Help information.
* E to exit.

In Text window:

* C to go to Calls window.
* D to go to Data window.
* F to go to Flow window.
* M to go to Main window.
* + to display next 10 lines.
* - to display previous 10 lines.
* N to display next line.
* P to display previous line.
* L nr to display line with specified number.
* G to continue execution after breakpoint.
* H to display Help information.
* E to exit.

In Flow window and in Data window:

* C to go to Calls window.
* D to go to Data window.
* T to go to Text window.
* M to go to Main window.
* F to go to Flow window.
* G to continue execution after breakpoint.
* H to display Help information.
* E to exit.

In Calls window :

* D to go to Data window.
* T to go to Text window.
* F to go to Flow window.
* M to go to Main window.
* R nr to select specified routine level.
* G to continue execution after breakpoint.
* H to display Help information.
* E to exit.

13.2.2 Breakpoints

Breakpoints can be set, removed or listed with the aid of three functions:

- BRKPT
- UNBRK
- LSTBRK

BRKPT(Name, Line, command) will set a breakpoint in the specified NODAL function and NODAL line at the specified NODAL command.

Example

```
2.10 SET A=1; SET B=2; SET C=3

>BRKPT(MAIN, 2.1 , 2)
```

willset a breakpoint in the MAIN program at line 2.1 at the second command. Which means that execution will be interrupted when the second command SET B=2 is going to be executed.

UNBRK(Name,Line,command) will take away the breakpoint set at the specified function, line and command.

LSTBRK will list all currently set breakpoints.

The maximum number of breakpoints will be 10.

The breakpoints can only be set and removed in interactive mode.

CHAPTER 14

APPENDIX A : NODAL errors

14 APPENDIX A : NODAL errors

1	Illegal line number
2	Illegal format specification
3	Illegal arithmetic expression
4	Ambiguous command
5	Illegal delimiter
6	Attempt to divide by zero
7	Working area full
8	Nonexistent name
9	Wrong variable type
10	Resources exhausted
11	Command not properly terminated
12	Unallocated error
13	Nonexistent line addressed
14	Illegal shuffle attempted
15	Error in IF command
16	Escape typed
17	Illegal edit command
18	Illegal ask command
19	Error in erase command
20	Argument list error
21	File error
22	Error in save command
23	Array dimension error
24	Square root of negative number
25	Illegal arctangent arguments
26	Sine argument too big
27	Cosine argument too big
28	Power error , negative argument
29	Power underflow
30	Exponential argument too big
31	Logarithm argument <= 0
32	Device not connected
33	Unauthorised action
34	Hardware error
35	Illegal equipment number
36	Illegal property
37	Value out of range
38	Not implemented command
39	No such computer
40	Result string filled
41	Syntax error
42	No such file
43	File already exists
44	No file space
45	Link not open
46	Remitted data lost
47	End of file
48	Equipment error
49	SIOM error
50	Illegal error number
51	Checksum error
52	In-line function area full

53 Syntax error in DEFINE command
54 Error in $SET command
55 String function failure
56 Illegal concatenation
57 Error in $IF command
58 Error in $ASK command
59 String expected
60 Pattern too big
61 Bad pattern match
62 Bad pattern
63 Bad pattern assignment

Both error 41 and error 48 will have an additional error message allocated to them, to explain the individual error sources in more detail. For the equipment this additional error message is completely determined by the error messages of the involved equipment.

CHAPTER 15

APPENDIX B : NODAL in EBNF

15 APPENDIX B : NODAL in EBNF

```
NODAL program line =
 line-digit [digit] . line-digit [digit] NODAL line
 line-digit = ('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')
 digit = ('0'| line-digit)
 NODAL line = NODAL statement { ';' NODAL line} |
                [ NODAL line ';'] '%' {character}
 NODAL statement =
 [ NODAL command <command syntax> | <function call> ]
 NODAL name = capital { capital | digit | : | . }
 NODAL statement = askcom | callcom | dimcom | defcom |
 docom | editcom | endcom | erasecom | execcom | forcom |
 gotocom | ifcom | imexcom | listcom | loadcom | oldcom |
 opencom | overcom | quitcom | remitcom | returncom |
 rofcom | runcom | savecom | setcom | typecom | valuecom |
 waitcom | whilecom | dollarask | dollardo | dollarif |
 dollarmatch | dollarpat | dolarset | dollarvalue |
 oncom | offcom

 askcom        = 'A'|'AS'|'ASK'
 callcom       = 'C'|'CA'|'CAL'|'CALL'
 dimcom        = 'DI'|'DIM'|'DIME'|'DIMEN'|'DIMENS'
 defcom        = 'DE'|'DEF'|'DEFI'|'DEFIN'|'DEFINE'
 docom         = 'DO'
 editcom       = 'ED'|'EDI'|'EDIT'
 endcom        = 'EN'|'END'
 erasecom      = 'ER'|'ERA'|'ERAS'|'ERASE'
 execcom       = 'EX'|'EXE'|'EXEC'|'EXECU'|'EXECUTE'
 forcom        = 'F'|'FO'|'FOR'
 gotocom       = 'G'|'GO'|'GOT'|GOTO'
 ifcom         = 'IF'
 imexcom       = 'IM'|'IME'|'IMEX'
 listcom       = 'LI'|'LIS'|'LIST'
 loadcom       = 'LO'|'LOA'|'LOAD'
 oldcom        = 'OL'|'OLD'
 opencom       = 'OP'|'OPE'|'OPEN'
 overcom       = 'OV'|'OVE'|'OVER'|'OVERL'|'OVERLA'
 quitcom       = 'Q'|'QU'|'QUI'|'QUIT'
 remitcom      = 'REM'|'REMI'|'REMIT'
 returncom     = 'RET'|'RETU'|'RETUR'|'RETURN'
 rofcom        = 'RO'|'ROF'
 runcom        = 'RU'|'RUN'
 savecom       = 'SA'|'SAV'|'SAVE'
 setcom        = 'SE'|'SET'
 typecom       = 'T'|'TY'|'TYP'|'TYPE'
 valuecom      = 'V'|'VA'|'VAL'|VALU'|'VALUE'
 waitcom       = 'WA'|'WAI'|'WAIT'
 whilecom      = 'WH'|'WHI'|'WHIL'|'WHILE'
 dollarask     = '$A'|'$AS'|'$ASK'
 dollardo      = '$D'|'$DO'
 dollarif      = '$I'|'$IF'
 dollarmatch   = '$M'|'$MA'|'$MAT'|'$MATC'|'$MATCH'
```

```
dollarpat    = '$P'|'$PA'|'$PAT'|'$PATT'|'$PATTE'
dollarset    = '$S'|'$SE'|'$SET'
dollarvalue  = '$V'|'$VA'|'$VAL'|'$VALU'|'$VALUE'
oncom        = '?ON'
offcom       = '?OF'|'?OFF'

number       = ['+'|'-'] integer | real | hexa | octal
integer      = digit { digit }
real         = digit { digit } '.' {digit} [ scalefactor ]
scalefactor  = 'E' ['+'|'-'] digit {digit}
octal        = '[' octal-digit { octal-digit}
octal-digit  = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'
hexa         = '[[' hexa-digit { hexa-digit }
hexa-digit   = digit|'A'|'B'|'C'|'D'|'E'|'F'
string = '"'{ character }'"'|"'" { character } "'"

DIMENS statement = dimcom ['-' int |'-' str] identifier
            '('expression [',' expression]')'
int = 'I'|'IN'|'INT'|'INTE'|'INTEG'|'INTEGE'|'INTEGER'
str = 'S'|'ST'|'STR'|'STRI'|'STRIN'|'STRING'
ERASE statement = erasecom { identifier | group | line }
SET statement = setcom identifier'=' expression
Expression = ["-"|"+"] operand { operator operand}
operator   = "-"|"+"|"/"|"*"|"↑"
operand    = '(' Expression ')' | number | identifier
DO statement = docom  expression {'!' expression} ['!']
END statement = endcom
RETURN statement = returncom
GOTO statement = gotocom  expression
WHILE statement =
       whilecom expression relation expression
         { 'OR' expression relation expression }
relation = '<'|'<='|'<>'|'='|'>'|'>='
FOR statement = forcom  identifier '='
                 expression [, expression] [, expression ]
ROF statement = rofcom
IF statement = logical IF | arithmetic IF
logical IF =
 ifcom expression relation expression
{ 'OR' expression relation expression }
arithmetic IF = ifcom '(' expression ')'
            expression [',' expression] [',' expression]

$SET statement = dollarset identifier '=' concatenation
concatenation = (string | [format] expression |
                 identifier | control) [concatenation]
format = '%' expression | '%,' | ']' | ']]' | '?'
control = '&' expression | '\' expression | '!'
identifier = {'$'} NODAL name
$DO statement = dollardo concatenation
$IF statement = logical $IF | arithmetic $IF
logical $IF =
dollarif concatenation relation concatenation
  { 'OR' concatenation relation concatenation }
arithmetic $IF = dollarif '('identifier {identifier} '-'
                          concatenation ')'
          expression [',' expression ] [',' expression ]
```

```
$PATTE statement = dollarpat identifier "=" pattern
pattern = simplepattern ['.'NODAL name|'$'NODAL name]
                  [ pattern ]
simplepattern = (NODAL name|string) {NODAL name|string}|
                '('pattern')'|NODAL name '!'NODAL name
$MATCH statement = dollarmat
          ( string|identifier | '<' concatenation '>' )
          ( pattern [ '=' concatenation ] )
          [ ':' expression ]

TYPE statement = typecom concatenation {','concatenation}
ASK statement = askcom [string] identifier
                     { [string ] identifier }
$ASK statement = dollarask [string] identifier
                        { [string] identifier }
LIST statement = listcom {identifier|expression|listid}
listid = 'ALLP' | 'ALLV' | 'ALL' | 'ALLD' | 'ALLR'
CALL statement = calcom identifier [ '(' paramlist ')']
paramlist = param {, param}

DEFINE statement = defcom '-' ( call | fun | str )
                        identifier [ '(' paramdefs ')' ]
call = 'C'|'CA'|'CAL'|'CALL'
fun  = 'F'|'FU'|'FUN'|'FUNC'
str  = 'S'|'ST'|'STR'|'STRI'|'STRIN'|'STRING'
paramdefs = funparam { ',' funparam }
funparam = ( 'V' | 'R' | 'S' ) '-' identifier
VALUE statement = valcom expression
$VALUE statement = dolarval concatenation
OPEN statement = opencom identifier

SAVE statement = savecom  filename
             {line | group | identifier | saveid}
saveid = 'ALLP' | 'ALLV' | 'ALLD' | 'ALL'
LOAD statement = loadcom filename
OLD statement  = oldcom filename
RUN statement  = runcom [ '[' line ']' ] [ filename ]
OVERLA statement = overcom filename [ paramspec]
paramspec = expression {(' '|',' expression }

IMEX statement = imexcom '(' comp ')' NODAL statements
EXECUT statement = execcom '(' comp ')'   ( line | group )
                       { identifier | line | group }
REMIT statement = remitcom identifier { identifier }
WAIT statement =
waitcom ('(' comp ')'|'-T' expression|'-C' expression)
HELP statement = helpcom
?ON  statement = oncom
?OFF statement = offcom
```

Index