THE NODAL SYSTEM FOR THE SPS

M.C. Crowley-Milling and G.C. Shering

GENEVA
1978
Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

Copyright CERN, Genève, 1978

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use.

If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.
ABSTRACT

A comprehensive description is given of the NODAL system used for computer control of the CERN Super-Proton Synchrotron. Details are given of NODAL, a high-level programming language based on FOCAL and SNOBOL4, designed for interactive use. It is shown how this interpretive language is used with a network of computers and how it can be extended by adding machine-code modules. The report updates and replaces an earlier one published in 1974.
## CONTENTS

1. Introduction ........................................... 1

2. Commands and Variables ............................... 2
   2.1 Simple Commands TYPE and SET .................. 2
   2.2 NODAL Names and Variables .................... 2
   2.3 Numbers .......................................... 3
   2.4 Expressions ...................................... 4
   2.5 Strings .......................................... 5
   2.6 Arrays ........................................... 5
   2.7 Lines with Multiple Commands .................. 7
   2.8 Abbreviation of Commands ....................... 8

3. Programs in NODAL ..................................... 9
   3.1 Line Numbers ..................................... 9
   3.2 The LIST Command ................................. 9
   3.3 Error Reporting .................................. 10
   3.4 The DO Command and Error Handling ............. 11
   3.5 The RETURN Command .............................. 13
   3.6 The GOTO Command ............................... 13
   3.7 The END Command ................................. 13
   3.8 The % Command .................................. 14
   3.9 ESCAPE and the QUIT Command ..................... 14

4. Decisions and Loops .................................. 15
   4.1 The IF Command .................................. 15
   4.2 The $IF Command ................................ 16
   4.3 The FOR Command ................................ 17
   4.4 The WHILE Command ................................ 18

5. Editing and Debugging ................................ 19
   5.1 Line Editing ..................................... 19
   5.2 The EDIT Command ................................ 20
   5.3 The ERASE Command .............................. 20
   5.4 Debugging ........................................ 21

6. System Variables ..................................... 22
   6.1 Read and Write Functions ......................... 22
   6.2 Mathematical Functions .......................... 23
   6.3 Call Functions ................................... 23
   6.4 Function Parameters ................................ 24
   6.5 Data Modules ..................................... 25
   6.6 Equipment Functions ................................ 27
   6.7 Data Module Support Functions .................. 27
   6.8 Data Tables ....................................... 28
   6.9 Data Module Protection .......................... 28
7. Multi-computer Programs ........................................... 31
   7.1 The IMEX Command ............................................. 31
   7.2 The EXECUTE Command .......................................... 31
   7.3 The REMIT Command ............................................ 32
   7.4 The WAIT Command ............................................ 32
   7.5 An Example .................................................... 33
   7.6 Parallel Processing ........................................... 33

8. Use of Files ...................................................... 35
   8.1 The Working Area .............................................. 35
   8.2 The RUN and OLD Commands ................................... 36
   8.3 Saving Programs and Data .................................... 37
   8.4 The LOAD Command ............................................. 37
   8.5 Task Common, the ARGs, and STRARG ......................... 38
   8.6 The OVERLAY Command ......................................... 39

9. More About Input and Output .................................... 41
   9.1 The TYPE Command ............................................. 41
   9.2 The ASK Command .............................................. 42
   9.3 The $ASK Command ............................................. 43
   9.4 The IDEV and ODEV Streams .................................. 44
   9.5 Alternative Input and Output Streams ....................... 44

10. More About Strings ............................................... 46
    10.1 Name Indirection ........................................... 46
    10.2 String Functions ........................................... 47
    10.3 Concatenations ............................................. 48
    10.4 The $DO Command ............................................ 49
    10.5 The $MATCH and $PATTERN Commands ......................... 49
    10.6 Patterns ................................................... 51
    10.7 Pattern Functions .......................................... 52
    10.8 Self Modifying Programs ................................... 54

11. Defined Functions ............................................... 55
    11.1 The DEFINE Command ....................................... 55
    11.2 The OPEN Command .......................................... 57
    11.3 The VALUE and $VALUE Commands, ERROR Function .......... 57
    11.4 The Defined Function Area ................................... 58
    11.5 Utility Commands ........................................... 58
    11.6 Recursion .................................................. 59
    11.7 Other Uses of the Defined Function Area .................. 59
12. Real Time Facilities ........................................... 61
   12.1 Synchronising Interactive Programs ................. 61
   12.2 Scheduling Independent Programs .................. 62
   12.3 Some Implementation Details ....................... 63

13. Writing Machine Code Modules for NODAL ............... 64
   13.1 The Header ............................................ 64
   13.2 Parameters and Data Modules ....................... 66
   13.3 Parameters for General Purpose Functions .......... 67
   13.4 String Descriptors .................................. 68
   13.5 Working Space ........................................ 69
   13.6 Support Macros ...................................... 70
   13.7 Other Types of Module ................................ 71

14. NODAL Element Lists ....................................... 72
    14.1 Element Types ....................................... 73

APPENDICES

1. Line Edit Commands ....................................... 80
2. Error Definitions ........................................ 81
3. Example of Machine Code Module ....................... 83
4. NODAL Element Types ................................... 84
5. Function Documentation .................................. 85
6. Command Summary .......................................... 95
1. Introduction

This report describes the NODAL system for the SPS as used during 1978. The report describes the NODAL language itself, how it is used interactively, and how it can be extended by adding machine code modules.

The NODAL language is a high level programming language with two special features. The first is a syntax which supports multi-computer network programming. The second is extensive string handling features which are required to support good programs for operator interaction. The language is based on FOCAL and SNOBOL4, with some influence from BASIC.

NODAL is an interpretive language. It is designed for interactive use and this is reflected in its structure. A prime aim of NODAL is to permit programming by hardware and accelerator specialists rather than by professional programmers. The SPS control system consists of about 30 computers linked together by a message transfer system. NODAL does not hide the network from the user, but rather opens it up to him for manipulation in his program.

Some of the examples in this report are specific to the SPS accelerator. These are not essential, however, and NODAL is being used in fields as remote as psychology and neuro-physiology. On the other hand the report does not describe everything even for an SPS computer. The most important features in any given application are covered by the special functions added to NODAL. A good example is that of the SPS consoles. These special functions and their use are described in the report on the consoles.

Chapter 2 gives an introduction to NODAL commands, then the rest of the report describes how these commands can be built into programs. An important feature of an interpreter is that all commands can be used in the "immediate" mode, so that they are obeyed immediately they are typed in. This is very useful for investigating or recovering from fault situations which are inadequately catered for in the normal NODAL program.

This report replaces and brings up to date the previous report "THE NODAL SYSTEM FOR THE SPS - 1974", LAB II-CO/74-2, December 1, 1974. It incorporates improvements in the presentation of the report, and also the improvements to NODAL itself which have evolved from three years use in the construction, commissioning, and operation of the SPS.
2. Commands and Variables

NODAL is an interactive interpretive system and so is used by typing directly on-line at the terminal, either a Teletype or a VDU (cathode ray tube display plus keyboard). The smallest useful element of NODAL is the command. Commands can be typed on the terminal and NODAL will obey them when the "return" key is depressed. NODAL signifies that it is ready to accept commands by writing the character ">" at the beginning of the line. If this character is not there, or does not appear when the "escape" key is depressed, then you must get an expert to help. The actions required to get NODAL running depend on the particular operating system being used.

2.1 Simple commands TYPE and SET

In order to get information from NODAL the "TYPE" command can be used, for example

```
TYPE BCT(3)
```

is a command which might write on the screen the value of the third circulating beam current reading.

```
SET INJPHS=12
```

is another command which will set the R.F. phase at injection to 12 degrees. Commands consist of a command key word, TYPE and SET in the above, followed by the command body, whose syntax varies from command to command.

A summary of all commands is given in Appendix 6.

2.2 NODAL Names and Variables

A NODAL name consists of a string of up to six characters (no spaces allowed). The character set is A-Z, 1-9, plus the colon ":" and period ".". A name must begin with a letter. Examples of names are

```
A      ABC     PS.5      VB:INJ
```

Two types of variables are distinguished by their names; program variables and system variables. Program variables can be created automatically by the SET and ASK commands. For example the command

```
SET A=1
```
would create a program variable A if it did not already exist. Program variables can only contain more than two characters if they contain a period.

System variables must contain at least three characters. They can contain a colon for mnemonic purposes. Thus

A B1 I.MAX SIGN.I

are program variables, whereas

ABC P1A INJ:VB PS:1

are system variable names and so can never be created by the SET or ASK commands.

2.3 Numbers

A number in NODAL is normally an unbroken sequence of digits containing only one decimal point. Examples of numbers are

1 .1 6.25

Numbers are stored to an accuracy of about nine decimal digits, though printouts are rounded to six digits unless otherwise requested.

Some other special cases of number representations are accepted in addition to the above. Examples of these are

1.3E-4

This means 1.3 times ten raised to the power minus four. Numbers with decimal exponents of up to ±4000 can be stored, though only exponents of up to ±2400 can be converted from character strings. A difference from FORTRAN is that "E-4" is not allowed, but "1E-4" is.

NODAL does all calculations in floating point form and there are no special rules or restrictions for integers. This means that integer data can be manipulated with no worry about scaling. As most of the accelerator input and output is done on the basis of 16 bit integers, NODAL provides several ways of representing such integers.
Octal and Hexadecimal Integers

Integers in the range -32768 to 32767 can be represented by a string of characters preceded by "[" to denote octal representation, and by "[[" to denote hexadecimal representation. For example

```
[177777
[[FFFF
```
can both be used to represent the integer -1.

RADIX36 Integers

Integers in the range -32768 to 32767 can also be represented by a RADIX36 string of the characters 0-9 (values as normal), and A to Z (values 10 to 35). If a string is preceded by a "#", for example

```
#ABC    #A1    #123
```
it is taken as a RADIX36 integer in NODAL. Each place in the character string is worth 36 times the place to the right, instead of 10 times as in decimal and eight as in octal. The algorithm used provides a positive integer, modulo 2 to the power 16, which is then taken as a signed integer by NODAL.

2.4 Expressions

These are combinations of numbers, variables and mathematical operators which can be evaluated by the interpreter to give a numerical result, e.g. the expression

```
10^3*3/10+21-2
```
can be evaluated to give the number 319. Arithmetic operations are performed from left to right except that exponentiation (^) is done first, then multiplication (*), then division (/), then subtraction (-), then addition (+). Thus 6+6*2 is evaluated as 18 as multiplication is done before addition. Brackets can be used to alter the order of operations, e.g. (6+6)*2 will give 24. Expressions can contain numbers, program variables and system variables, e.g.

```
TYPE 2*K*BCT(3)
```
or

```
SET INJPHS=A+2*B
```

A difference from FORTRAN is worth noting here. Multiplication has a higher priority than division in NODAL, not the same priority as in FORTRAN. Thus
1/2*PIE*I = 1/(2*PIE*I)

and

A/B*C = A/(B*C) not (A/B)*C

The fact that NODAL does not have an explicit number type of integer leads to the restriction that a negative number cannot be raised to any power, even if the power is represented by an integer.

The numeric expression is one of the basic syntactic entities of NODAL and can in general be used anywhere a number is required. In all examples where a number is used it can be replaced by any expression, unless otherwise mentioned explicitly.

2.5 Strings

NODAL has two data types: numbers as discussed above, and strings. A string is often a sequence of visible characters but is formally any sequence of 8 bit bytes. The maximum length of a string in NODAL is 80 bytes. Within this limit strings are handled and stored dynamically and the length is automatically taken care of. The command

$SET A = "THIS IS A STRING"

creates the string variable A with the contents consisting of the character string which was enclosed by the quotes. If it is desired that the character string should contain double quotes then single quotes can be used in the $SET command to delimit the string. For example

$SET A = "FRED'S FRIEND SAID"

$SET B = '"WHERE ARE YOU"

creates two strings A and B such that the TYPE command

TYPE A B

will result in the type-out

FRED'S FRIEND SAID "WHERE ARE YOU"

2.6 Arrays

Up to present we have only considered simple variables, such as "A" in the above, which contain a single number or string. Arrays of numbers or strings can also be used.
Arrays are created by means of the DIMENSION command, for example

```
DIM A(36)
```

will create an array of three word floating point numbers which can be accessed as A(1) .. A(36). Array elements are initialised to zero when dimensioned. Integer arrays can also be created, for example:

```
DIM-INT A(X)
```

would create an integer array X words long. Integer arrays are only used to reduce storage requirements. All calculations are done in floating point and rounded, not truncated, to the nearest integer before storage in the array. The array elements are immediately reconverted to floating point format on retrieval from the array.

Two dimensional arrays of both types can be created, e.g.

```
DIM B(10,2)
DI-I J(4,5)
```

Arrays are stored, as in NORD FORTRAN IV, in ascending order of storage location. Two dimensional arrays are stored by column, i.e. the first subscript varies most rapidly. For instance the array A(3,2) is stored as

```
A(1,1)
A(2,1)
A(3,1)
A(1,2)
A(2,2)
A(3,2)
```

A two dimensional array can be accessed as a single dimensional array in a program, in the above example A(4) = A(1,2). A check is made on array boundaries to prevent writing outside the array by programming error.

String arrays can also be used. They must be created first using the DIMENSION command, for example

```
DIM-STR A
```

which creates a string array A. Only single dimensional string arrays are available. The number of elements is not declared in the DIMENSION-STRING command. Elements can be inserted using the $SET command, for example

```
$SET A(5) = "GEORGE"
```
It is not necessary that the elements A(1) to A(4) exist, for example one could then say

\[ \$SET \ A(10) = "JOE" \]

The string array A now contains two elements A(5) and A(10). A useful function is FIND, which returns the array index of a string in an array, or -1 if the string is not in the array. For example

\[ \text{TYPE} \ \text{FIND(A,"JOE")} \]

would cause the number 10 to be printed out.

Variables and arrays in NODAL have an existence which is independent of the command or program which created them. They exist as little packets of information with a header giving the name, type, dimensionality, etc., and can be moved about between computers or to and from peripheral devices. If an array is redimensioned, or obtained from some external source, then any pre-existing array or variable of the same name is first deleted. A useful function is ARSIZE which returns the number of elements in an array. For example

\[ \text{FOR I=1,ARSIZE(A); T A(I)!} \]

is a command which will type out all the elements of a numeric array "A" and

\[ \$SET \ \text{STR}(\text{ARSIZE} \ \text{STR}+1)="NEW \ ELEMENT" \]

is a command which will add a new element to a sequentially numbered string array "STR".

2.7 Lines with Multiple Commands

So far we have considered only single commands. NODAL will accept multiple commands separated by semi-colons on a single line. No action is taken until the "RETURN" key is pressed. Then all the commands are obeyed one after the other. For example

\[ \text{SET A=1; SET B=2; TYPE SIN(A+B)/SIN(A-B)} \]

The line of commands is a very important concept in NODAL. The end of the line marks the end of the scope of commands such as FOR, WHILE and IF. For example

\[ \text{DIM A(10); FOR I=1,10; SET A(I)=I} \]
\[ \text{WHILE CAMAC(1,1,0,0)=0; SET CAMAC(1,2,0,16)=1} \]
\[ \text{IF MAGNET(4)>300; SET MAGNET(5)=0} \]
These commands will be discussed in more detail later.

2.8 Abbreviation of Commands

NODAL commands can be abbreviated provided no ambiguity results. For example "DIMENSION" is usually abbreviated to at least "DIM". Typing "HELP" on the keyboard will give a list of the NODAL commands together with the minimum permissible abbreviation.
3. Programs in NODAL

Up to this point only commands which are executed immediately by NODAL have been discussed. As the NODAL system is interpretive, these commands are understood and obeyed immediately without any compilation process. If the line of commands is prefixed by a line number, however, the line is not executed immediately but is stored for later execution, usually as part of a sequence of commands.

3.1 Line Numbers

Line numbers must be in the range from 1.01 to 99.99 . The numbers 1.00, 2.00, etc., are illegal line numbers as they are used to identify the entire group. The number to the left of the point is called the group number.

1.10 SET A=1
1.30 SET B=2
1.50 TYPE A+B

The stored program can be executed by typing "RUN". Once it has been debugged it can be saved on a file and executed using the RUN command, or at the occurrence of an interrupt, or at a given time, or at regular intervals of time.

Lines are automatically sorted by NODAL into order of increasing line number. For instance if a line 1.2 is now typed after the above, it will be inserted between lines 1.1 and 1.3. Often one only types line numbers as 1.1, 1.2, etc. but NODAL stores and lists them as 1.10, 1.20 etc. and line 1.15 would be put between lines 1.1 and 1.2.

Lines can be a maximum of 80 characters long, including the line number.

3.2 The LIST Command

The command "LIST" on its own causes NODAL to list the entire program on the terminal. "LIST 1.1" or "LIST 2", however, will cause only line 1.1 or group 2 to be listed. "LIST 1.1 2 3" causes line 1.1 and groups two and three to be listed. The parameters of the list command must be line or group numbers, not expressions.
3.3 Error Reporting

One of the features of NODAL is its detection of program errors at run-time, together with advanced facilities for error tracing and subsequent program editing.

When an error occurs during execution of a command an error message is printed, for example

```
DEVICE NOT CONNECTED AT LINE 2.2
```

NODAL then stops and enters command mode so that the faulty line can be corrected. As an example consider the following sequence of operations

```
>1.1 SET A=2; TYPE "A" A !
>1.2 SET B=4; TYPE "B" B !
>1.3 TYPE AB+B
>RUN

A 2
B 4
NONEXISTENT NAME AT LINE 1.30
```

Note that the ">" is typed by NODAL to indicate that it is expecting a line or immediate command to be typed.

In long lines the user can get a better indication of where the error occurred by typing CTRL/B (holding down the CTRL key, then hitting the B). NODAL then prints out the offending line with an arrow indicating where the error occurred, e.g. in the above example CTRL/B would give

```
1.30 TYPE AB+B
   ^
```

NODAL is now automatically in EDIT mode ready to change line 1.3. The editing facilities will be described in more detail later.
3.4 The DO Command and Error Handling

The DO command allows separate groups of the program to be done as subroutines, with control returning automatically to the command following the DO command. For example

```
1.10 SET A=1; DO 10
1.20 SET A=2; DO 10; END
10.10 TYPE "COSINE OF" A "IS" COS(A)!
10.20 TYPE " SINE OF" A "IS" SIN(A)!
```

A single line can also be done as a subroutine, for instance

```
DO 10.1
```

The group structure of NODAL is useful for dividing a program into logical sub-units or sub-routines. Each such sub-unit can be given a different group number then executed using the DO command. The DO command would normally be given in a program sequence, e.g.

```
1.1 FOR INJPHS = 5,15; DO 2
1.2 END
2.1 TYPE INJPHS
2.2 TYPE BCT(3)
```

The DO command can also be given directly by the key-board so that parts of a program can be tested individually. This is useful as many programs consist of an acquisition part, a calculation part, then a control part.

The line number field in the DO command can be any valid expression. For example

```
DO BUTTON+1
```

might cause group 4 to be done if button 3 is touched.

In computer control systems errors have a quite different scope than in normal computer programming. A program may run very satisfactorily one day but may run into error on the next because of some hardware error, or a device not being connected, or because it is being used by someone not authorised to use the particular hardware in question. The normal result of such an error is described in section 3.3 above, i.e. the program stops with the appropriate plain language error message and awaits operator intervention. It is possible, however, to use programmed intervention in order to take corrective action, some alternative action, or even just to display details of the error on some special device such as a colour screen. This is accomplished using the DO command, for example
DO 10!11

will normally be identical to "DO 10". If group 10 encounters an error, however, group 11 will be done instead of the program stopping, and the program will continue after the DO command as if no error had occurred. The command

   DO 10!10!12!98

will attempt group 10, if it fails group 10 will be tried again, if it again fails group 12 will be attempted, and if that fails group 98 will be done.

   DO 10!

will do group 10 up to the point that any error occurs then return, but the program will continue as if no error had occurred.

In order to pass on information about errors NODAL has the system variable ERROR which is set to zero at the beginning of a DO command, and which takes on the error number if an error occurs in the group. This can be used to act only on specific errors, for example

   1.10 DO 10!11
   1.20 END

   10.10 SET MAGNET(6,#CUR)=217

   11.10 IF ERROR<>32; SET ERROR=ERROR
   11.20 SET MAGNET(6,#CON)=1; DO 10

is a simple program which will set the current in magnet number six to 217 amps by doing group 10. If MAGNET returns an error, however, group 11 will be done. If the error is not number 32, which means "DEVICE NOT CONNECTED" then group 11 will be provoked into error by setting "ERROR" equal to the value it received from group 10. If the error is 32 then line 11.2 will connect the magnet and re-do group 10.

A useful string function is ERMES which returns the string corresponding to the error number. Error numbers can range from 1 to 127. The meaning of the error numbers can be printed out using the command

   FOR I=1,127; TYPE I ERMES(I)!

The error messages may vary from system to system, but the usual ones for error numbers 1 to 63 are given in appendix 2.
3.5 The RETURN Command

Normally a DO subroutine executes all the lines in the group and returns when there are no more lines to do. Often, however, early termination of the group is required, say as a result of testing some condition. This is achieved using the RETURN command. When a RETURN command is encountered during execution of a DO subroutine, the program exits from its subroutine status and returns to the command following the DO command that initiated the subroutine status.

It is not possible to jump out of a DO group using either the GOTO or IF commands described below. Any such attempt will be treated as a RETURN command. DO subroutines themselves, however, can be nested to any depth. Thus eventually control will always return to the command following the DO command, unless some termination command such as END is encountered first.

3.6 The GOTO Command

The GOTO command causes NODAL to transfer control to a specific line in a running program. After executing the specified line, NODAL continues to the next higher line. The GOTO causes a program branch; i.e. a jump to a previous or subsequent line, e.g.

GOTO 1.3

The line number may be replaced by any valid NODAL expression. This gives a computed GOTO, e.g.

GOTO 4-2.7

gives the same result as GOTO 1.3.

Care should be exercised if the target of a GOTO is outside the current group. This is only allowable if the groups are part of the main program and never called as DO subroutines.

3.7 The END Command

A program normally ends when it has reached the end of the highest numbered line. Earlier ending can be provoked using the END command. In the on-line mode END causes control to be returned to the on-line terminal. In real-time programs END causes a return to monitor and the program disappears.
Common practice is to concentrate the main logic of the program into the lower numbered groups, often into group 1, then use the higher numbered groups, say 10 to 99, as DO subroutines. The END command is then required at the end of the main logic groups to prevent the program continuing on to the subroutine groups in sequence.

3.8 The % Command

Beginning a command string with the character % will cause the remainder of that line to be ignored so that comments may be inserted into the program. Such lines will be skipped over when the program is executed, but will be typed out by a LIST command.

3.9 ESCAPE and the QUIT Command

When a program is taking some unwanted action, or is looping endlessly, it can be stopped by pressing the "ESCAPE" key. In on-line NODAL systems this usually stops the program and NODAL goes into command mode awaiting instructions from the operator. In the background SINTRAN-III systems, however, "ESCAPE" gives control to SINTRAN-III. Return to NODAL is obtained by typing "CONTINUE".

The QUIT command stops NODAL and puts the terminal back into control of the operating system. In background SINTRAN-III this is the same as pressing "ESCAPE".
4. Decisions and Loops

The ability of a program to make decisions is what enables it to exhibit "intelligence". An automatic facility for looping permits complex repetitive operations to be expressed in a concise manner. This chapter describes the NODAL commands which realise these facilities.

4.1 The IF Command

The ability to make decisions is one of the most important features of a computer program. In NODAL this is done by using the IF, $IF, or WHILE commands.

The IF command allows the execution of the remainder of a line of commands to be conditional on the validity of a logical expression, e.g.

IF A<132; TYPE "LESS THAN"; DO 4

If the condition is true, here A less than 132, the rest of the line is executed. Otherwise the rest of the line is ignored. The logical expression takes the form of two arithmetic expressions separated by the logical operators

> greater than
< less than
= equal to
>= greater than or equal to
<= less than or equal to
<> not equal to

The first arithmetic expression must not be enclosed in brackets to avoid confusion with the branching IF which will be described later. The two expressions are considered equal if the difference between them is less than $5E-8 relative to the first.

The condition may be the "OR" of several logical expressions, for example

IF A=1 OR A=3 OR A=5; SET A=A+10; SET B=LOG(A)

An effective "AND" condition is obtained by cascading several IF commands, for example

IF A=1; IF B=2; SET C=3

A three branch version of the IF command is also available. This allows program jumps to be controlled by the sign of an arithmetic expression. For example:
IF (A-10)3.2; DO 2

If the value of the bracketed expression is negative, here if A is less than 10, then control is transferred to line 3.2. Otherwise the next command, here DO 2, is executed.

IF (X)3.2,3.3; DO 2

In this example, if X is negative, control is passed to 3.2, if X is zero, control is passed to 3.3, if X is positive, the next command, here DO 2, is executed.

IF (X)3.1,3.2,3.3

In this example a branch is always made, to 3.1, 3.2 or 3.3 according to whether X is negative, zero or positive.

The expression in brackets is taken to be zero if its magnitude is inside the range ±1E-6.

4.2 The $IF Command

This command can be used to compare two strings. For example

$IF A="YES" OR A="PERHAPS"; DO 4

which will DO group 4 if the string variable A contains "YES" or "PERHAPS". The operators ">", "<", etc. can be used just as for the IF command described above. The lexical comparison is made character by character along the string on the basis of the ASCII values of the characters. In particular

NULL<0<9<A<Z

The ASCII value of any character can be obtained on-line using the ASCII function which takes a string as a parameter. For example ASCII("A") would return the value 65.

A branching version of the $IF is also available exactly as described above for the numeric IF command. For example

$IF (A="YES") 2.1, 2.2, 2.3

which compares the contents of the string variable A with the constant string "YES" and branches according to the comparison. The comparison is made lexically between the two concatenations separated by the minus sign. Thus if A is lexically before "YES" control goes to 2.1, if A is identical to "YES" then control goes to 2.2, and if A is lexically after "YES" control goes to 2.3. The branching can be reduced exactly as explained for the arithmetic IF command.
4.3 The FOR Command

The FOR command uses the end of the line to define its field of action, e.g.

```
FOR I=10,20; SET VACVALV(I)=1
```

will open the vacuum valves 10 to 20 inclusive. System variables can also be used as FOR variables, e.g.

```
FOR INJPHS=6,18; TYPE INJPHS, BCT(3) !
```

will cause a table of injection RF phase and circulating beam current to be typed. The ! causes a new line to be taken. The FOR command in general takes three expressions, for example

```
FOR I = X+2 , Y/10 , 100; DO 4
```

where I starts with the value of the first expression, increments by the value of the second expression, as long as it has not reached, or is equal to the third. As shown in earlier examples the second expression can be omitted, in which case an increment of +1 is assumed. All three expressions can be fractional or negative as desired, provided the step is of the sign to take the variable from the starting value through the ending value.

The expressions are evaluated only once at the start of the FOR command and the loop variable is always reset to its correct value at the start of each loop. For example

```
FOR X=1,4; TYPE X; SET X=10
```

will type all values of X from 1 to 4. Early exit from a FOR loop can be made using the GOTO command, for example

```
1.20 FOR I=1,10; TYPE I; IF I>5; GOTO 1.7
```

The ROF command is specially designed for breaking out of FOR loops. For example the immediate command

```
FOR I=1,10; FOR J=1,10; T I J I*J!; IF I*J>50; ROF
```

will stop with I=6 and J=9. In program sequences ROF is equivalent to GOTO the next line after the line containing the associated FOR command (or series of FOR commands). It is particularly useful in DO groups, for example

```
1.10 FOR I=1,10; FOR J=1,10; DO 10
1.20 ---------
----
1.90 END
```
If a ROF command is encountered in group 10, for example after an IF command, then group 10 terminates and control is returned to line 1.2.

4.4 The WHILE Command

This is similar to the IF command above except that the rest of the line is executed in a loop until the condition is no longer true. For example

```
WHILE CAMAC(0,1,0,0)=0; SET CAMAC(0,2,0,16)=1
```

causes 1 to be written repeatedly into module 2 until module 1 returns 1. The condition is evaluated and checked for truth before each execution of the rest of the line. The condition can have "OR" clauses just as in the IF command above. The command "WH 0=0" is often used to give continuous repetition of some commands until the "ESCAPE" key is pressed.
5. Editing and Debugging

One of the features of NODAL is that it contains its own editing, debugging, and run-time system so that separate edit, compile, link, run, and debug phases of program development do not require separate processors and associated files.

5.1 Line Editing

The line editing facility of NODAL is identical to that used by the QED editor program. In essence, some already existing line of text, called the old line, is mapped onto a new line being created. Normally the old line is the last line entered. Special characters are used to control the mapping process. These are the control characters, designated as for example, CTRL/C which is control C. These characters are obtained by holding down the "CTRL" key then typing the required character.

The most obvious mapping commands are the copy commands, for example CTRL/C copies one character from the old line onto the new line. CTRL/O Character, where Character is any single character, copies all characters up to but not including Character. The editor is normally in the replace mode so any character other than a CTRL character is entered in the new line. For example if one has just typed

\[
1.1 \text{ LET } A=1
\]

which should be

\[
1.1 \text{ SET } A=1
\]

one can type CTRL/O L which copies up to but not including L, then type S which replaces L, then CTRL/D which copies the rest of the old line and terminates the EDIT.

It is also possible to insert characters at any point by typing CTRL/E. NODAL types < then inserts all characters typed until another CTRL/E is typed which is echoed as >. For example suppose we have typed

\[
1.1 \text{ SET } A=1
\]

and we want

\[
1.1 \text{ SET } A123=1
\]

we can type CTRL/Z A which copies the old line up to and including A, then CTRL/E to insert, then 123, then CTRL/E again to end the insert. CTRL/D copies the rest of the old line to finish the edit. It is also possible to skip over unwanted characters in the old line, e.g. CTRL/S skips one character in the old line. CTRL/X C skips up to and including the character C. When editing long lines one can forget what has been done previously. CTRL/Y copies in the rest of the old
line and restarts the edit. If one wants to see what is there, one can type CTRL/H which copies and types out up to the end of the line. Typing CTRL/Y again restarts the edit on this clearly typed out line.

When typing a new line or editing an old line mistakes can be made. Typing CTRL/A deletes the last character on the new line and echoes ^ . CTRL/W deletes the last word and echoes \. CTRL/Q deletes the whole new line, echoes back arrow, then gives a carriage return line feed so that one can start again. This latter need not be used for lines with a line number, however, as typing carriage return ends the edit and inserts the new line however faulty. This line then automatically becomes the old line and so can be edited and corrected provided the line number is not changed.

Other control characters are available for other editing functions, a complete list being given in appendix 1.

5.2 The EDIT Command

The simplest way of editing a line is to retype it. In this case NODAL inserts the new line in place of the old line with the same number.

The EDIT command, however, enables one to use the line editing facilities described above on an already existing program line. For example

EDIT 10.5

will select line 10.5 as the old line and enter line edit mode. One can then use the line edit control characters to change the line. The line number can even be edited. In this case a new line with the new line number is entered and the old line is left in. A useful variant is

EDIT 10.5,L

The ",L" causes the line to be listed out first, before edit mode is entered.

5.3 The ERASE Command

Lines, groups, and variables may be erased from the user list by means of the ERASE command, for example:

ERASE 1.1 2 A B
will erase line 1.1, group 2 and the data elements A and B.
The command

ERASE ALL

clears user store and should be used before typing a second
program after typing a first so as to avoid a mixup between
the two. The command "ERASE ALLP" will erase all the program
but leave the variables, and the command "ERASE ALLV" will
erase all variables but leave all the lines of program.

The ERASE command can be used in a program to recover working
space used by a large array, say, or some initialisation
group. Any lines erased must have higher line numbers than the
line containing the ERASE command, and higher than any active
line in the program at the time the ERASE command is executed.

5.4 Debugging

A simple method of debugging is to put in some "TYPE" commands
at strategic points in the program, taking advantage of the
on-line editing capability of NODAL. One can also stop the
program by pressing the "ESCAPE" key when NODAL indicates
where it stopped, for example

ESCAPE TYPED AT LINE 10.5

The names and values of the variables at this point can be
listed by typing "LISV".

Giving the command ?ON will cause each line of a program to be
typed before it is executed so that the user can follow the
progress of his program. ?OFF disables this facility.
6. System Variables

Much of the power and flexibility of an interpretive control system comes from its "system variables". These are names which are built into the system ready for use by NODAL at any time. Arrays and variables with a specific meaning can be built in, but usually system variable names refer to functions which will perform some service for the user.

A list of the system variables in any specific NODAL system can be obtained by typing "LISR". The resulting output gives the names, parameter types, sizes, etc. for each variable or function. Further description of the functions can be obtained from the appropriate manual. An excerpt is shown in appendix 5. This covers some of the more commonly used functions.

6.1 Read and Write Functions

These allow processed data, often from external hardware, to be handled in the same way as simple program variables. For example

\[
\text{SET } Z=2*\text{CAMAC}(1,1,0,0)
\]

will set the program variable Z equal to twice the value returned by the system variable CAMAC. Values can also be written out via system variables, for example

\[
\text{SET } \text{CAMAC}(1,1,0,16)=-1
\]

System variables can be read only, write only, or read-write. CAMAC is a read-write function, so one can write

\[
\text{SET } \text{CAMAC}(1,1,0,16)=2*\text{CAMAC}(1,1,0,0)
\]

String functions can also be read only, write only, or read-write. For example the command

\[
\$\text{SET LEGEND(0)=}\text{DATE}
\]

will read the date and time then write it out in ISO standard form on the top line legend on the console touch screen. "DATE" is a read only string function which returns the date and time as a character string, and "LEGEND" is a write only string function which creates touch button legends.
6.2 Mathematical Functions

These have the usual meaning and are really just read only functions of a mathematical rather than accelerator type. For instance

\[ \text{SET } Y = \text{SIN}(X) \]

has an obvious meaning. The functions available are

<table>
<thead>
<tr>
<th>Function name</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIN(X)</td>
<td>Sine of X, X in radians</td>
</tr>
<tr>
<td>COS(X)</td>
<td>Cosine of X, X in radians</td>
</tr>
<tr>
<td>AT2(Y,X)</td>
<td>Arctangent of Y/X in range -PIE to PIE</td>
</tr>
<tr>
<td>SQR(X)</td>
<td>Square root of X</td>
</tr>
<tr>
<td>EXP(X)</td>
<td>Exponential of X</td>
</tr>
<tr>
<td>LOG(X)</td>
<td>Natural logarithm of X (LOGe(X))</td>
</tr>
<tr>
<td>ABS(X)</td>
<td>Absolute value of X</td>
</tr>
<tr>
<td>INT(X)</td>
<td>The integer part of X</td>
</tr>
<tr>
<td>FPT(X)</td>
<td>The fractional part of X (same sign as X so that ( \text{INT}(X) + \text{FPT}(X) = X ))</td>
</tr>
<tr>
<td>SGN(X)</td>
<td>(\pm 1) according to sign of X</td>
</tr>
<tr>
<td>MOD(X,Y)</td>
<td>X modulo Y</td>
</tr>
<tr>
<td>PIE</td>
<td>3.14159265</td>
</tr>
</tbody>
</table>

6.3 Call Functions

These are invoked using the CALL command. For example

\[ \text{CALL LISR} \]

will list all the resident system variable names, together with their types and parameter specifications. The keyword "CALL" in the above is optional so that the above command can be abbreviated to

\[ \text{LISR} \]
This makes it look just like a command. Function names, however, must be typed out in full and cannot be abbreviated. Parameters can be used to pass information to or from the function. For example

\[ \text{FFT}(A,-1) \]

will do a forward Fast Fourier Transform on the array A.

6.4 Function Parameters

Parameters can be of three types: numeric value, string, or reference. The parameter type is determined by the function and is checked at the time of function activation. For general purpose functions the parameters are evaluated by NODAL before the function is called. The type of each parameter can be read from the LISR output. In the so-called "free-functions" and in string functions the parameters are evaluated by the function itself. The function documentation must be consulted to get the parameter types.

Numeric value parameters are used to pass a number to the function. In the calling sequence any valid expression can be used.

String parameters are used to pass a string to the function. In the calling sequence a constant string (sequence of characters enclosed in quotes) can be used, or a string variable, or a string array element. In some free functions and string functions a "concatenation" (to be defined later) can be used, as indicated in the function documentation.

Reference parameters are used to pass the address of a NODAL element, for example a variable or an array, to the function. The function can then access this element. For numeric elements the function can read from the element, write to the element, or manipulate its contents. Strings, however, can only be read.

An example of a function using all three parameter types might be

\[ \text{CALL NAME}(3, \text{"YES"}, \text{AR}) \]
6.5 Data Modules

These are a type of function especially suited for controlling the SPS or other process equipment. They are based on the existence of a number of equipments of the same type, where each equipment can be described by a number of "properties". One data module provides the control and acquisition for all equipments of that particular type. The data module has two essential parameters. The first is the equipment number which identifies the particular equipment in the series of the same type. The second is the "property" of that equipment which is to be controlled. For example one data module could do all the work for all the sputter ion pumps, say

\[ VPS(N,P) \]

where \( N \) is the pump number, \( P \) is the property of the pump under consideration. So

\[ \text{TYPE VPS}(40,#CUR) \]

might access the current in pump 40

\[ \text{SET VPS}(40,#SWI) = 1 \]

might switch pump 40 on. Data modules may work in two ways: firstly they may access the hardware directly, secondly they may merely access tables which are used or filled by an autonomous task. It is even possible to combine both, for instance

\[ \text{TYPE VPS}(40,#CUR) \]

could access the true pump current, whereas

\[ \text{TYPE VPS}(40,#CUS) \]

might access the last scan value. The property specification can be omitted, for example

\[ \text{TYPE VPS}(40) \]

In this case a property of zero is handed to the data module which should interpret this as the default or normal property.

Property specifications are handed to the data module as an integer. This integer is usually chosen to have a convenient mnemonic in RADIX36 form as shown in the examples above. The property can of course be stored and passed in a variable. Arithmetic can be done on property specifications where this is meaningful. For example the command

\[ \text{FOR P} = \#DL1,\#DL4; \text{ TYPE VPS}(3,P) \]
would type out the properties #DL1, #DL2, #DL3, and #DL4 of equipment number 3. Numerically related properties can only be used conveniently with the last digit in the range 0 to 9 due to the RADIX36 convention.

Some data module properties require transmission of arrays, for example to load a function generator, or to read statistics from a beam current transformer. This is obtained by using the data module in the CALL mode, for example

```
DEVICE(AR,"W",3,#PRO)
```

could be used to transmit the array "AR" to unit 3 of the "DEVICE" using the property "#PRO". Similarly

```
BCT(AR,"R",2,#LG1)
```

would read the logging information indicated by the property "#LG1" from unit 2 into the array "AR".

When an error occurs the data module signals this back to the NODAL interpreter by returning an error code whose value describes the error. This can be left for the interpreter to handle as described in section 3.3, when the program is stopped with an error message. Another possibility is for the programmer to handle the error using the "DO!" feature of NODAL as described in section 3.4. This is the recommended method for mainline operation programs where only summary error handling is desired. For more detailed hardware maintenance and fault finding programs, an additional facility is provided in the data module call. An optional parameter after the property specification can be supplied. If this is supplied and there is an error then the interpreter sets the error number into the place specified by the optional parameter, then goes on to the next command. It is up to the programmer to deal with the error if the returned code was non-zero. For example

```
SET VPS(40,#SWI,FL)=1
```

might attempt to switch pump 40 on. If, for example, there was no power on pump 40 then the appropriate error code would be set into the variable "FL" and the program would continue. Zero is set into "FL" if there is no error. The third parameter can be any NODAL settable entity. This optional error parameter is a particularity of data modules and the equipment functions described below. It is not available for any other type of function.
6.6 Equipment Functions

An "equipment function" is a special case of a data module where the name refers to only one equipment. Thus a unit number need not, and in fact must not, be used in the calling sequence. For example

\[
\text{SET INJ:VB(#CUR)=456}
\]

could set the current in the unique magnet called INJ:VB to 456 amps. As before the property specification can be omitted when the command becomes simply

\[
\text{SET INJ:VB=456}
\]

assuming that the default property of INJ:VB is #CUR. Array handling facilities are equally available to equipment functions. Again no unit number specification is required, for example

\[
\text{PAD(AR,"W",#ZON)}
\]

writes the contents of array "AR" to the equipment function "PAD", as required by the property "#ZON".

6.7 Data Module Support Functions

Data modules provide the basic low level access routines to the hardware. They can be called directly from NODAL for maximum flexibility. It is also possible to write higher level functions which come between NODAL and the data modules to provide more powerful facilities for special cases.

Two special support functions have been provided for fast multiple access to data modules. These are DPREA and DPWRT. These permit a single command to be used say to read all the magnets in a beam line, or all the wires in a particular beam monitor. Examples of their use are

\[
\text{DPREA(AUXPS,7,#CUR,AR)}
\]
\[
\text{DPWRT(AUXPS,7,#CUR,AR)}
\]

These access the data module "AUXPS", starting at unit 7 and using the property "#CUR" to read or write data from the array "AR". The number of values transferred depends on the size of the array or the number of equipments, whichever is less.
Another useful support function is LISDM. This provides some on-line documentation on a data module by looking at its internal structure. For example

\[
\text{LISDM(BCT)}
\]

will list on the user terminal all the properties of BCT, together with some other useful information.

6.8 Data Tables

Data modules are so named because they contain a data table. Collectively the data modules form the on-line data base for the SPS. The data table holds parameters such as the hardware address, calibration factors, tolerances, limit values, etc., together with a record of the desired values and statuses. Each of these items can be read or changed using the appropriate property. Normal control and acquisition actions are processed by the data module in accordance with the parameters in the data table.

A data table is organised in rows and columns. Each row corresponds to a separate equipment, and the number of columns depends on the equipment type. The data modules with their data tables are held in the satellite computer to which the hardware is connected. Three support functions are provided to help with back-up and reload of data tables as a whole. These are DTSIZE which returns the size of the data table in terms of 16 bit words, and DTREA and DTWRT which permit the entire data table to be read from or written to an array. For example

\[
\text{DTREA(AUXPS,AR)}
\]

will copy the entire data table of the data module "AUXPS" to the array "AR".

6.9 Data Module Protection

A NODAL user has access to all the hardware via the data modules. To prevent mis-use, either by error or intent, a protection system is used. Equipment can be operated either through programs or by immediate commands, but all actions go through the data module. Protection is therefore applied in the data module with respect to the user.
Associated with each user are two entities: a section number, and a capability word (16 bits). A casual user has section zero and capability zero and most data modules will only permit read accesses. A user can log-in, however, when he gets his correct section number and capability word. Logging-in can be done in two ways: firstly by typing, using a secret unechoed password known only to the person himself; secondly by inserting a personal identity card into the special reader provided on each console and pressing the "RESET" button. The section number and capability word are arrived at by mutual agreement between the person involved and those responsible for the system.

Broadly speaking, a person's section indicates which equipment he is responsible for and hence he can change properties such as hardware address, limits, etc. The capability word is used to indicate permission to operate equipment using the normal operational properties of the data module. The capability word is divided into 16 separate bits, each indicating a particular class of equipment. A person is only allowed to operate the equipment if he has the appropriate bit set to one. Typically the operations personnel have many capability bits set to one so that they can operate most equipment, but could well have no ability to modify limits, etc. as they are not in any hardware section.

Data modules apply the protection in a way which can be different for each property. The user's section and capability are examined in the light of the property protection code. The access may even depend on the particular equipment in the data module. The protection codes are

0 - No check
1 - Data module capability required
2 - Data module section required
3 - Not used
4 - Unit specific capability required
5 - Unit specific section required
6 - 1 or 5
7 - property specific section required

The protection only applies to the write action of the property. No check means that anyone can use the property. A capability requirement means that the user's capability word must have a specified bit set in order to use that property. A section requirement means that the property can only be used by someone belonging to the specified section.
Protection codes 1 and 2 cover the whole data module, that is the capability or section requirement is the same for all units in the data module. This is the normal case for a single user data module. Some data modules are multi-user, for example the auxiliary power supply data module. Such data modules can use protection codes 4 and 5 which require an extra word in the data table to specify the capability or section requirement for that particular unit.

Protection code 6 is a special one for use in the experimental areas. Here we want equipment to be operable by either the experimental area operators (all equipment), or by the particular experimental group to which that equipment has been allocated (a small subset for each experiment).

Protection code 7 allows a property to be allocated to a section other than the owners of the data module.

Facilities are provided in data modules to switch protection off, and to take units out of service.
7. Multi-computer Programs

The syntax of NODAL allows a program to be expressed as a number of separate tasks which can execute in different computers in the network. Facilities are provided to transfer data between these tasks in a controlled fashion. This is one of the most important features of NODAL for the SPS Computer Control System, and is handled by the four commands IMEX, EXECUTE, REMIT, and WAIT.

7.1 The IMEX Command

IMEX, for immediate execute, causes a command to be obeyed in a remote computer and any result to be sent back to the originating computer for type out. For example

IMEX (5) TYPE BCT(3)

will cause the command "TYPE BCT(3)" to be sent to computer 5 for immediate execution. The result, say 59.6, will be sent back to the originating computer and typed out. All the remainder of the line after the "IMEX (5)" is sent down, so more than one command can be transmitted. A typical command might be

IM(INJ)FOR I=1,6; T AUXPS(I,#STA)!

Thus any immediate command line which could be typed in on a terminal attached directly to the remote computer concerned can, with the simple "IMEX" prefix, be used at any other terminal on the network. The section and capability protection system is carried over the links along with the command.

The computer specification enclosed in the brackets depends on the particular operating system in use. It is sometimes a number, a name, or either a number or name. A number can always be represented by any valid expression.

7.2 The EXECUTE Command

This is the main command for programmed interaction. An example might be

EXECUTE (GP3) 2 AR

will send group 2 and the array "AR" to the computer "GP3" for execution as a remote sub-task of the program. The operation of this command depends on the fact that NODAL elements are relocatable and computer independent. Also NODAL lines are not processed until run-time, so they can be entered and stored in
computers in which they are illegal provided they are sent to another computer where they can be executed.

Both groups and single lines can be sent for execution. Also any number of any type of NODAL named elements can be sent as data. These named elements must be from those contained in the "working area". The concept of the working area will be discussed later. It is the area whose contents are listed by the LISV command. For example we can use

\[ \text{EXEC}(5) ~ 2 ~ 3.1 ~ \text{AR} ~ X ~ Y ~ I \]

A specialised variant of the EXECUTE command is EXECUTE-PRIORITY. This permits small time-critical tasks to be executed at a high priority. The treatment of this command depends on the receiving computer. Some treat it in exactly the same way as an ordinary EXECUTE command, others give it higher priority but a very small working area for execution.

### 7.3 The REMIT Command

Often programs are sent to another computer in order to get information back. This is handled by the REMIT command, for example

\[ \text{REMIT} ~ A ~ B \]

sends back the arrays (say) A and B, presumably after they have been filled with information. REMIT means send back and so is only meaningful in a task sent from one computer to another using the EXECUTE command. The destination of the REMIT command is implicitly the main NODAL task which did the EXECUTE. The REMIT command is the last command executed in the EXECUTE task, i.e. it contains an implied END.

Even if the sub-task does not contain a REMIT command, or encounters an error before reaching the REMIT command, NODAL does an effective "REMIT" at the end. This sends either an "OK" message or the error information.

### 7.4 The WAIT Command

The WAIT command in its form

\[ \text{WAIT} ~ (GP3) \]

causes data REMITted from an EXECUTE task to be read in. The program waits until the data is there. If there is an error in the EXECUTE program, this is flagged at the time of the WAIT. If the program does not contain an explicit WAIT command then
NODAL does an effective "WAIT" at the end of the program, or before allowing another "EXECUTE" to go out to the same computer. It is at this time that any errors will be detected.

Remitted data is read in at the time the WAIT command is obeyed. The variables or arrays are then ready for use. It is not necessary to create or dimension them beforehand. If the names already exist in the working area they will be deleted as the new ones are read in. During the period between the EXECUTE and WAIT commands the program can do other things. In particular it can work on previously remitted data with no fear that this will be overwritten by new remitted data as this cannot happen until the WAIT command is executed. If the remitted data is ready before the WAIT command is executed it is stored in the "pipe-line" ready for the WAIT.

7.5 An Example

The following is a program which might be executed in a console computer to get a scan of the ejected beam position at entry to a septum. It shows the use of the EXECUTE, REMIT and WAIT commands.

1.1 ASK "INITIAL POSITION=" IP "FINAL POSITION=" FP
1.2 SET P=IP; EXECUTE (8) 3.2 P
1.3 FOR P=IP+1,FP; DO 2
1.4 END

2.1 WAIT-CYCLE 4;EXECUTE (8) 3 P; WAIT (8)
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

7.6 Parallel Processing

A program often needs data from several computers in the system. This can sometimes take some time, for instance when analogue to digital conversions are required. A considerable increase in speed can be achieved by executing the necessary sub-tasks in parallel in the different computers. The following program shows an example of this. It acquires 108 values of the horizontal beam position from the hypothetical data module "HPOS" in six satellite computers, combines them into a single array, then sends this array to the display computer for display on one of the screens.
1.1 WAIT-CYCLE 6
1.2 FOR I=1,6; EXECUTE (I) 2
1.3 DIMENSION A(108)
1.4 FOR I=1,6; WAIT (I); FOR J=1,18; SET A(18*(I-1)+J)=B(J)
1.5 EXECUTE(DISP) 3 A
1.6 GOTO 1.1

2.1 DIM B(18)
2.2 FOR I=1,18; SET B(I)=HPOS(I)
2.3 REMIT B

3.1 SET ODEV=15; CLEAR; HISTO(A)
8. Use of Files

One of the features of NODAL is that programs and data are essentially relocatable and computer independent. This allows the full flexibility of a general purpose file system to be used for program storage, overlays, etc. In the following "FILENAME" represents the name of an arbitrary file on the file system in use. The file systems in use at the SPS are the SYNTRON file system on the on-line computers, and the NORD file system on the TSS computer. The appropriate manual should be consulted to get full specifications of "FILENAME" and the associated commands to list, create and delete files, etc.

8.1 The Working Area

When a program is typed in, the lines are stored in the "working area". This can be pictured diagrammatically as follows

```
Lines of Program
Variables and Arrays
Stack
```

First are stored the lines of program, arranged in numerical order. If a new line is typed in then the existing lines are "shuffled" in order to put this new line in its correct position in the list.

After the lines come the named data elements. These are arranged in order of creation. Each data element contains a header giving the name, type, and dimensions if it is an array. If a DIMENSION command specifies a name which already exists in the working area, then the element of this name is deleted and the new one added to the end of the list.
The NODAL interpreter consists of re-entrant code and there is a separate working area for each task. The interpreter uses a stack to store its internal local variables. This stack starts at the end of the working area and grows up towards the last data element. The space between the top of the stack and the end of the data elements is free space. When this runs out NODAL gives the error message "WORKING AREA FULL".

8.2 The RUN and OLD Commands

Once a program has been typed in it can be started by typing "RUN". This starts the program at the lowest numbered line. Execution can also be started at some higher line, for instance

RUN [2.7]

will cause execution to start at line 2.7. Programs stored on files are also started using the RUN command, for example

RUN FILENAME

will bring the contents of "FILENAME" into the working area and start execution at the lowest numbered line. Any previous contents of the working area are first deleted. The RUN command can be used as part of a program, so one program can "RUN" another. These programs can also be started at line numbers other than the first, for instance

RUN [5] FILENAME

will run "FILENAME" starting at the lowest numbered line of group 5. The "5" can of course be replaced by an expression indicating any valid line or group number.

Often we wish to get a program from a file into the working area just to look at it, or to modify it temporarily before running it. This is accomplished using the OLD command, for example

OLD FILENAME

clears the working area, brings in the contents of "FILENAME", then returns control to the interactive user. Both OLD and RUN are the last commands executed on their line.
8.3 Saving Programs and Data

Programs and data can be stored for later reference using the SAVE command, e.g.

```
SAVE FILENAME
```

will save the current program on the file "FILENAME". Data can also be saved, for example

```
SAVE FILENAME AR
```

will save the array "AR" on the file "FILENAME". Only the data element "AR" will be saved, not the program. The command

```
SAVE FILENAME ALLP AR
```

will save all the program plus the array "AR" on "FILENAME". Several parts of a program and several data elements can be saved together, for example

```
SAVE FILENAME 2 LEGS PROGS
```

will save group 2 of the program, plus the arrays "LEGS" and "PROGS" on the file called "FILENAME". All the variables can be indicated by "ALLV", for example

```
SAVE FILENAME 2 ALLV
```

will save group 2 plus all the variables. All the contents of a file, both lines and data elements, are brought into the working area by the RUN and OLD commands.

8.4 The LOAD Command

The LOAD command is mainly used to load data from a file into the working area in the course of execution of a program. For example

```
LOAD FILENAME
```

will load the contents of "FILENAME" into the working area. If a data element to be loaded has the same name as an existing element then the existing element is automatically deleted.
If the file contains program lines these are also loaded, and replace any lines of the same number in the existing program. This is done on a line basis, however, so that if a group is loaded then it will be merged with the existing group, keeping lines from the existing group which are not present in the new group. If this is not wanted the old group can first be erased.

If the LOAD command is used to load lines or groups during the execution of a program, then any lines affected must have higher line numbers than any line currently active in the program.

8.5 Task Common, the ARGS and STRARG

It is frequently convenient to divide a job into several files which can be run one after the other in a fixed or program-controlled sequence. This provides additional flexibility, eases modifications and editing, and overcomes the limit of working area on the size of the task to be performed. For instance we could store the program in three parts on the files say FILE1, FILE2, and FILE3. Then to start the program one types "RUN FILE1" which causes the first part of the program to be executed. This part ends in the command "RUN FILE2" which causes the second part to be loaded and executed, and so on.

It is usually necessary to exchange data between the separate parts. If a lot of data, e.g. an array ABC, is to be exchanged then another file must be used. The penultimate command in FILE1 might then be "SAVE FILE4 ABC" which would put the array ABC onto FILE4. Then the first command in FILE 2 could be "LOAD FILE4" so that the array ABC would then be loaded exactly as it was. In the consoles a special fast scratch file "BEEP" is available to the main interactive user for this purpose.

Often, however, only a few values need to be communicated between programs. For this purpose 16 Task Globals called ARG(1)...ARG(16) have been provided. These retain their values as long as the particular NODAL task is active, i.e. are not affected by ERASE ALL, OLD, SAVE, LOAD etc. They are set or read as

\[
\text{SET ARG}(5) = X \\
\text{SET X} = \text{ARG}(5)
\]

This allows sequences of chained programs to be easily set up, the main interchange parameters being held in ARG(1) ARG(16). A string can also be passed in this way using "STRARG". The first program can write into STRARG with say
$SET STRARG="FILEFRED"

and the next program can read its contents with the command

$SET S=STRARG

Only one STRARG is provided, and it can only hold up to 40 characters. It is particularly useful for passing file names. The ARG's are specific to the working area in use. Any other task, including a sub-task sent to a remote computer for execution, has its own set of ARG's. Thus a remote sub-task has no access to the ARG's of the sending task.

8.5 The OVERLAY Command

The OVERLAY command allows a subprogram stored on a file to be loaded and executed in the middle of a main program without disturbing the main program at all. The subprogram can use the same line numbers or variables as the main program with no interference. The subprogram overlay temporarily shares the working area with the main program so their sum of their lengths must fit into the working area. The subprogram disappears after use, however, so the main program can then call other overlays. Overlays can be nested to any depth permitted by the size of the working area. Thus the line, say 5.7 in some program,

5.7 TYPE MBBH(1) ; OVERLAY FILENAME ; TYPE MBBH(1)

will cause the value of MBBH(1) to be typed out, then some program stored on FILENAME to be executed, then MBBH(1) to be typed out again.

Communication between the main program and the overlay can be achieved using the Task Globals ARG(1)... ARG(16) and STRARG as described above. This is such a useful facility that it is catered for automatically in the overlay command, e.g.

OVERLAY FILENAME 2,3,4

This command runs the overlay FILENAME as before, but first sets ARG(1) = 2, ARG(2) = 3 and ARG(3) = 4. These values can be picked up and used as desired in FILENAME. Only the first 8 ARG's can be set automatically in this way. Any legal expressions, separated by commas, can be used to specify the ARG's.
A disadvantage of overlays is that error reporting is always done at the level of the main program. Although the error message always describes the error that occurred in the overlay, the line number and the position indicated by the CTRL/B key give the place in the main program where the overlay is called. Overlays can of course be tested interactively by pre-setting the ARG's and running the overlay as a main program. This gets rid of syntax and logical errors. To avoid run-time errors it is advisable to restrict the use of accelerator hardware in overlays. Overlays are particularly useful for creating fixed parts of displays.
9. More About Input and Output

In many programs, particularly those involving displays and operator interaction, much of the code concerns input and output. It is important that these operations be easy and convenient. This chapter describes the facilities offered by NODAL.

9.1 The TYPE Command

This command has been mentioned earlier and has been further illustrated in several of the examples. It is the most commonly used facility for presenting information to the operator. An example might be

TYPE "VALUE IS" A B ! "END"

If "A" is a string variable containing "FRED =", and "B" is a numeric variable of value 2.5362, then this could result in the type-out

VALUE IS FRED = 2.5362
END

The sequence of items after the "TYPE" keyword can be expressions for which the value is typed, strings including string variables and string array elements, and control items such as the "!" in the above which causes a new line to be taken.

Control sequences cause NODAL to take special action. The control sequences are as follows, where X represents any legal expression

! Take new line (carriage return plus line feed)
%X Change format
%, Use E format, field length of 17
\X Type ASCII equivalent of X
&X Type X spaces
]X Type out X in octal
]]X Type out X in hexadecimal
?X Type out X in binary

Normal fixed point for numeric values is 6 digits before the decimal point and four digits after, i.e. a total field of 11. This format can be changed using the % control character. The control sequence %8.03 changes the format to 3 digits after the point in a field 8 characters long. %, specifies E format, e.g. "TYPE %, 1" gives the type-out "1.0000000E0". Note that this use of % occurs inside the TYPE command, so does not cause confusion with the % command (comment). If the number is too large to fit into the specified format then E format is
used automatically. Integer format is obtained if % is followed by an integer. For example

```
TYPE %5 A+B
```

will type out the value of A+B as an integer right justified in a field of 5. The format specifier remains valid up to the end of the TYPE command being executed, unless changed later in the command. It is reset to its default value of %11.04 at the beginning of each new TYPE command.

Commas can be used to separate elements of a TYPE list. For instance

```
is the same as
TYPE A, B
but
is not the same as
TYPE A -B
```

The "\X" control sequence is widely used to control the printing or display device. It allows any 8 bit value to be sent to the device. Certain values are used to control the device, especially those from 0 to 31. For example "TYPE \12" will cause the line printer to take a fresh page, or will erase and reset a whole display screen. Other control characters can be used to select colour, line or column position, etc. The use of these control characters is discussed in more detail in the report on the consoles.

A single TYPE command can output a maximum of 80 characters. One can output longer lines, say to the 132 character line-printer, by using two TYPE commands one after the other and putting the "!" control character for new-line in the second TYPE command.

9.2 The ASK Command

The ASK command is normally used in programs to allow the user to type in data at specific points during the execution of his program. The ASK command is written in the form

```
11.99 ASK X Y Z(1)
```

When line 11.99 is encountered by NODAL a colon (:) is typed. The user can then type an expression, whose value will be given to the variable X. If only an asterisk (*) is typed the value of X is unchanged. The expression or asterisk is followed by carriage return, when the evaluation takes place. If more than one variable or array element is asked for, e.g.
X, Y, and Z(1) in the example above, then all three values can be typed on the one line separated by commas. Also text can be included in the ASK command. For example

11.99 ASK "VALUE OF X" X "VALUE OF Y" Y "VALUE OF Z" Z

When NODAL executes line 11.99, it first of all types

VALUE OF X:

The user can reply with, say, 2<carriage return>then 3<carriage return>then 4<carriage return>, which would look like

VALUE OF X : 2
VALUE OF Y : 3
VALUE OF Z : 4

The variables X, Y and Z now have the values 2, 3 and 4 respectively. If the user knows that Y and Z will be asked for directly after X he can type all three values at once, separated by commas, e.g.

VALUE OF X : 2, 3, 4

thus suppressing unnecessary printout.

9.3 The $ASK Command

This command allows strings to be input from the terminal. The $ASK command types a colon (":") then waits for the user to type a sequence of characters followed by carriage return. The sequence of characters is then assigned to the variable specified in the $ASK command. For example

$ASK "GOOD OR BAD" A

would type "GOOD OR BAD :" on the teletype then assign the typed in sequence to A. Several variables and array elements can be asked for in one $ASK command, just as in the numeric ASK command. The reply in request for each variable, however, must be terminated by pressing the return key, and cannot be separated by commas like the numeric ASK command. The prompting string, if any, will always be typed out. If the string variable or array element exists before the $ASK command is executed, then the previous contents is taken as the "old line" for editing purposes. Thus if one does not want to change the string one can type CTRL/D.
9.4 The IDEV and ODEV Streams

NODAL maintains two main input and output device streams called IDEV and ODEV respectively. For normal interactive work these streams are assigned to the terminal. This assignment can be changed for special purposes. For instance the command

```
SET IDEV=OPEN("R", "TERM-MODE")
```

will open the file TERM-MODE for reading and attach the interactive input stream to the output of this file. The stream will be reset to the standard input device when the new file is empty, unless of course some other assignment is processed first.

It is much more common, however, to re-assign the output device. This allows an immediate command or program to send the result of any TYPE command to an alternative destination. For example

```
SET ODEV=OPEN("W", "LINE-PRINTER"); LIST
```

will change the output stream so that the subsequent LIST command will cause the program to be listed on the line printer. The output stream is reset to the standard value at the end of a program, or just before return to command mode. Thus the prompting ">" is typed out on the correct device.

The OPEN function returns the "logical number" of the device or file, and reserves the device for this particular user. In some systems such as the console system, the device or display is ready for use and does not need to be reserved. Then one can use the logical number directly. For example

```
1.10 SET ODEV=10; T \12
```

might be the first line in a program and will direct all output to the main colour screen and will clear the screen.

9.5 Alternative Input and Output Streams

The logical number returned by the "OPEN" function in the above examples need not be used to set ODEV or IDEV, but can be stored for use in other ways. For example

```
SET Z=OPEN("A", "ACCUMULATING-FILE")
```

will open "ACCUMULATING-FILE" for append. The first parameter for the OPEN function can be "R", "W", or "A", for reading, writing, or appending. In the above example the logical number returned by the OPEN is stored in the variable "Z". The command
FOR I=1,ARSIZE(A); $SET OUTPUT(Z)=A(I)$

could be used to write the contents of the string array "A" to the file as a sequence of lines separated by carriage returns and line feeds. Such a file must be closed after use as NODAL has no automatic method of closing alternative streams. The CLOSE function is used for this, for example

CLOSE(Z)

will close the file whose logical number is stored in "Z". A line of text can be read from a file using the INPUT function, for example

$SET S=INPUT(Z)$

where "Z" contains the logical number of a file previously opened for read. A common way of using these functions is in programs which read text from one file, manipulate it with the advanced string processing facilities described later, then write the resulting strings back to some other file. The four main functions for reading and writing text to files are

INPUT  read a line from a file,
     and discard the line separation characters.
IMPC   read one character from a file
OUTPUT write a string to a file,
     and add the line separation characters.
OUTC   write a string to a file exactly as it is

Two numeric functions, INBT and OUTBT are also provided. These permit 8 bit bytes to be read and written as numbers. When writing, the number is converted to an integer and the lowest 8 bits written to the file. When reading from the file the byte is returned as a number in the range 0 to 255. These functions are mainly used in system-type applications for reading and modifying non-text files.
10. More About Strings

NODAL is not only an interactive language but is also, at least in the console computers, a language for programming interaction. For this the ability to handle character strings is very important. String variables, arrays, and functions have already been described, as have the $SET, $IF, and $ASK commands. In this chapter we treat the more advanced use of strings.

10.1 Name Indirection

Frequently the name of an item to be processed is not known at the time the program is written but depends on some choice made by the operator. This may be to type in the name in question, or select it from a list. Once the required name has been obtained, say in the string variable "S", the item can be accessed using name indirection. For example

\[ \text{DTREA($DM,$AR)} \]

might be used to read the data table of the data module whose name is stored in the string variable "DM" into the array whose name is stored in the string variable "AR". The name can also be stored in a string array element, for example

\[ \text{SET $S(5)(6,2)=10} \]

might set the item in row 6, column 2, of the numeric array whose name is stored in the 5th. element of the string array "S". Indirection can be nested to any depth, for example

\[ \text{SET $$S=3} \]

might set to 3 the numeric variable whose name is stored in the string variable whose name is stored in "S". Indirection can also be used for file names. For example

\[ \text{LOAD $S(1)} \]

might load the contents of the file whose name is stored in the first element of string array "S".
10.2 String Functions

We have seen the use of string functions for writing strings to and getting strings from external devices. String functions are also extensively used for manipulating strings within programs, for example

\[ \$SET B = \text{SUBS}(1,5,A) \]

sets the string variable B equal to the string returned by the string function \text{SUBS}. Here \text{SUBS} returns the substring 1 to 5 of the string variable A, i.e. the first five characters of A. A string contains up to 80 characters which are numbered 1 to 80 for the purposes of \text{SUBS}. The Nth. character in a string can be addressed as

\[ \$SET S=\text{SUBS}(N,N,A) \]

The function \text{SUBS} can also be used in write mode but cannot be used to expand strings.

String functions can cause the \$SET command to fail. For example in the line 3.1

\[ 3.1 \ \$SET A = \text{INPUT}(65):3.8; \ DO 4 \]

the \$SET command sets A to the string returned by the function \text{INPUT}. This is normally the string of characters read from file 65 up to the first carriage return line feed. If, however, there are no more characters left on the file, \text{INPUT} causes the \$SET command to fail, that is to GOTO line 3.8. If no failure return was specified ("3.8" in the above) the program would continue and A would be set to the null string. This special mechanism of failure should not be confused with normal errors which cause the NODAL program to stop. For example in the line 3.1 above a normal error would occur if file 65 was not opened for reading. This would cause the program to stop with a normal NODAL error message.

The functions which can be used to input and output strings from files were described in sections 9.4 and 9.5. Some other functions useful in string work are described below. The word "\text{CONC}" is used in the following examples to denote any valid concatenation as defined in the next section.

\[ \text{STRING} = \text{SUBS}(I,J,\text{CONC}) \]

returns substring I to J of concatenation

\[ \text{VALUE} = \text{SIZE}(\text{CONC}) \]

returns number of characters in concatenation

\[ \text{VALUE} = \text{EVAL}(\text{CONC}) \]

treats \text{CONC} as an arithmetic expression and returns its
MORE ABOUT STRINGS

VALUE = FIND(ARRAY,CONC) returns index of CONC in array, -1 if not found

STRING = ALPHA pre-defined string variable, returns A to Z

STRING = NUM pre-defined string variable, returns 0 to 9

10.3 Concatenations

In the above examples we have used constant strings and string functions to specify strings. These are special cases of concatenations. A concatenation is represented by a series of string defining elements which returns a string consisting of the individual strings joined end to end. For example

$SET A=" AND GEORGE"

$SET B="JOE" SUBS(1,5,A) "FRED"

would give the string B the contents "JOE AND FRED". A concatenation consists of a series of string elements, arithmetic expressions, or control sequences. The control sequences are exactly the same as those defined in section 9.1 for the TYPE command. Arithmetic expressions are converted to character strings again as in the TYPE command, using either the default format or a special format denoted by the % control sequence. Thus

$SET SESCAN(2)="CURRENT =" %3 KNOB

will write onto self-scan unit 2 a message say

CURRENT = 54

ie. the constant string "CURRENT =" followed by the value of the KNOB converted to a string according to the preceding format control %3. Elements in a concatenation can be separated by spaces but not by commas as is optional in the TYPE command.

Arbitrary characters can be written to strings using the "\" control sequence, again as described in section 9.1. This is frequently used in display work, for example

$SET A=\9 \I+7

will put two characters in the string variable "A". This defines, in the SPS display code, the column position on the
screen. Note that the expression "I+7" is evaluated at the
time the $SET command is executed and the resulting character
put into the string.

10.4 The $DO Command

The fact that NODAL is an interpreter allows another
interesting feature, the $DO command. For example

$DO A(5)

will take the contents of A(5) as an immediate command and do
it. The string is first copied into the $DO command before
being done, which allows constructions such as

10.1 $DO "ERASE 10.1; DEF-F NAME"

Thus first of all line 10.1 is erased then the rest of the
buffer is defined as the function NAME. The $DO command can be
used to construct immediate execute commands using the string
facilities to transmit calculated parameters, for example

FOR D=1,10; $DO "IMEX(LIB)DIR(" D ")"

will do repeated IMEX's to the computer "LIB" with the
parameter "D" replaced by the string representation of the
numbers 1 to 10 in the call to the function "DIR".

10.5 The $MATCH and $PATTERN Commands

These are very powerful commands for string manipulation and
are derived from the specialised string processing language
SNOBOL 4. An example is

2.7 $MATCH S(5) "JOHN" :2.1; DO 5

This matches the string in "S(5)" with the constant string
"JOHN". If the matching succeeds the next command, here "DO
5", is executed. If it fails the failure field, here ":2.1" is
used and in this case a branch is made to line 2.1. The
failure field is optional as in the $SET command. A more
general, though less used, form is for example

2.7 $MATCH <S1 S2> "JOHN" :2.1

where any valid concatenation contained between the angle
brackets can be matched.
The string or concatenation is matched against a "pattern". In the above examples the pattern was the constant string "JOHN". Patterns can either be "immediate" that is they are defined as they are used in the $MATCH command, or can be defined and stored prior to use by the $PATTERN command, for example

$PATTERN P1="JOHN"

this defines and stores in P1 the pattern on the right of the equals sign.

The $MATCH command can also contain a replacement field, for example

$MATCH S(5) P1="FRED"

would search the string in "S(5)" for the pattern "P1", and if found replace it with the string "FRED".

The concept of cursor position is an important one in many pattern functions. Normally a pattern match starts with the cursor in position zero, that is just to the left of the first character. An attempt is made to match the string with the pattern starting with the cursor in position zero, i.e. with the first character. If the match fails then the cursor is moved to position one and the match is tried starting from the second character. For example the command

$M "OYES" "YES"

first tries to match "OYE" with "YES" but it doesn't work, so it moves on one position where it finds a match. Matching in this way is inherently a time consuming process as the full string is matched for each character position, of which there can be up to 80. Frequently we only wish to try the pattern on the whole string starting with the first character, that is with the cursor in position zero. This can be ensured by including the pattern function "POS(0)" in the pattern. For example

$MATCH S POS(0) "YES"

will only match if "YES" makes up the first three characters in the string. Only one pass of the string is made so the result is obtained much more quickly.
10.6 Patterns

The simplest case of a pattern is a constant string. For instance the sequence

\[
\begin{align*}
2.1 & \text{ $\text{MATCH A } "YES" : 2.2; TYPE } "\text{INCLUDED}"; \text{ RETURN} \\
2.2 & \text{ TYPE } "\text{NOT INCLUDED}" \\
\end{align*}
\]

is a group which will search the contents of the string variable A for the pattern "YES". If it finds the pattern, eg. if A = "OYES" or "OH YES" or "YES OR NO" or "YESSIR", then it types "INCLUDED" otherwise "NOT INCLUDED".

Patterns can be more complicated than simple strings. They can include concatenation and alternation. Alternation is denoted by an exclamation mark ("!"), eg. the pattern

"YES" ! "NO"

will match either "YES" or "NO" and the pattern

"A" "B" !"C" "D"

will match either of the string "AB" or "CD", whereas

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

matches "ABD" or "ACD".

Patterns can contain assignment fields. The assignment field causes the part of the string which is matched to be copied into a variable. For instance

\[
\begin{align*}
$\text{PAT P1} &= "A" ("B" ! "C") "D" \\
$\text{FOR I = 1,10; } &A(1)$ \text{ P1 .OUTPUT} \\
\end{align*}
\]

The first command creates a pattern P1. The second scans the array A, matches each element against P1, if it is matched then the matched part either "ACD" or "ABD" is assigned to output. The dot (".") is called the conditional assignment operator. (Note it must be separated from "P1" in the above by a space as ":P1." is a valid NODAL name which is not the pattern P1).

Patterns can consist of several parts, for example

\[
$\text{PAT P3} = P1 .A1 P2 .A2$
\]

This creates a new pattern P3 which consists of P1 followed by P2. The match only succeeds if P1 immediately followed by P2 matches. If it succeeds, the string matched by P1 is copied into A1, and the string matched by P2 is copied into A2. If P1 P2 is not matched no copying is done. The "immediate assignment operator" dollar ("$") allows copying of partial
matches. For example the pattern P4

\$P4 \equiv P1 \ A1 \ P2 \ A2

matches in the same conditions as P3 above. However an attempted match by P4 might succeed insofar as P1 matches but fail overall because P2 doesn't match. The immediate assignment operator "$" works immediately P1 matches, without waiting to see if P2 is going to succeed. Pattern matching works by trying to match the pattern starting at the first character position. If no match here, it moves to the second character position and tries again, and so on. For instance if A is given by

\$A \equiv "YES, YES, YESNO"

then

\$MATCH A ("YES" \ $ \ OUTPUT \ "NO").OUTPUT

will produce the type-out

YES
YES
YES
YESNO

10.7 Pattern Functions

In the last section we considered only patterns made up of strings. Most of the interesting patterns use pattern functions. For instance

"A" \ ARB \ "D"

is a pattern that will match an "A" followed by any character up to and including "D". This is because ARB is a pattern function which matches any character.

\$POS(N) is a pattern function which only matches when the cursor is in position N. This "anchors" the pattern to occur at a fixed point in the string, often at the beginning. For instance

\$POS(0) \ "YES"

only matches if "YES" occurs at the beginning of the string.
RPOS(N) does the same as POS(N) but the position is counted from the end of the string, RPOS(0) being just to the right of the last character.

LEN(N) matches any string of length N. It is often used to check the length of a string before proceeding to further matching.

SPAN(CONC) matches the longest run of characters contained in CONC, starting from the current cursor position. For instance

POS(0) SPAN(ALPHA) . LB

could pick up an alphabetic label from the front of a string and assign it to the variable LB.

BREAK(CONC) is the opposite of SPAN as it matches up to but not including any break character contained in CONC.

ARB matches any number of any characters.

ANY(CONC) matches any single character contained in CONC.

NOTANY(CONC) matches any single character not contained in CONC.

TAB(I) matches up to position I.

RTAB(I) matches up to the I th position from the end of the string. For instance

TAB(0) RTAB(0)

always matches the whole of any string.

FAIL is pattern which, if tried, causes pattern mis-match and so causes other alternatives to be tried.

ABORT is a pattern which, if tried, causes the complete failure of the $MATCH command, i.e. no other alternatives are tried.
10.8 Self Modifying Programs

Another feature of an interpreter is that a program can modify itself during execution. This opens up the possibility of evolution in the software field. In practice this facility is mainly used to construct NODAL lines in computers benefitting from the powerful string processing facilities described above, then to send these lines down links for execution in computers with a simpler version of NODAL. This feature is implemented using the NODLIN function. This read-write string function is used as follows:

\[ \$SET \ S = \text{NODLIN}(10.1) \]

This puts the text of line 10.1, without the line number, into the string variable "S". NODLIN gives string function failure, as described in section 10.2, if the line does not exist.

\[ \$SET \ \text{NODLIN}(10.1) = S \]

This creates a line 10.1 by appending the contents of "S" to the line number 10.1. Any line number can be used, of course, but in making a line one must ensure that it has a higher number than any active line in order to avoid an "illegally attempted shuffle" of an active line as the new line is slotted into position.
11. Defined Functions

The interactive power of NODAL comes from its ability to call named functions. Up to present only subroutines and data modules built into the system have been discussed. This chapter describes how functions can be written in the NODAL language itself, then stored and loaded for future use.

In many languages for compile and run systems the named procedures appear as part of the main program listing. This is not so in NODAL as procedures can be used even without a program, i.e. in an immediate command. Thus defining a procedure is a separate exercise from writing a main program. All procedures, once defined, become external to the main program.

The console computers communicate with the hardware connected to the remote satellite computers by sending NODAL commands or programs over the data links and getting results REMITed back. To simulate the effect of simple immediate commands or programs in the console computer one can define NODAL named procedures which include the EXECUTE and REMIT commands. For instance the routine BCT in computer 5 can be simulated in the console computer by typing the sequence

```
1.1 EXECUTE(5) 1.2 ; WAIT(5) ; VALUE X
1.2 SET X = BCT(3) ; REMIT X
DEFINE-FUNCTION BCTR3
```

The immediate command "DEFINE-FUNCTION BCTR3" above creates NODAL defined function called BCTR3. The body of the function consists of the text in the buffer at the time the DEFINE command is given, here lines 1.1 and 1.2. The value of the function is given by the expression in the value command, the variable X in the above. The VALUE command also terminates the execution of the function, as it includes the END command.

11.1 The DEFINE Command

The DEFINE command turns the contents of the text buffer into a named procedure. Three types of procedure can be created, read/write numeric functions, CALL subroutines, and read/write string functions. These are differentiated as

\[
\text{DEFINE-FUNCTION NAME}
\]
\[
\text{DEFINE-CALL NAME}
\]
\[
\text{DEFINE-STRING NAME}
\]

The procedure can take parameters, e.g.
DEFINE-CALL NAME(V-ABC, R-DEF, S-GHI)

The names of the formal parameters in the above are ABC, DEF, GHI. The prefixes "V-", "R-" and "S-" denote the type of the formal parameter, either value, reference or string. These formal parameters are defined from the calling sequence when the routine is activated and so can be used in the body of the routine. For instance the simple function SUM can be defined as

1.1 VALUE ABC+DEF

DEF-FUN SUM(V-ABC, V-DEF)

and used, for instance, as "TYPE SUM(2,3)" which would type the result "5". Up to 8 formal parameters can be used, in any combination of types.

In some cases additional parameters are created by the system. In particular for DEF-FUN and DEF-STR an additional simple numeric variable FLAG is always produced. This has the value +1 if the function was used in the read mode, and -1 if the function was used in the write mode. Also if these types are used in the write mode the variable "VALUE" is created. This is a numeric read-only variable for a DEF-FUN type, and a string variable for a DEF-STR type. VALUE is given the value of the expression on the right hand side of the equals sign in the SET command, e.g.

1.1 EXECUTE (8) 1.2 VALUE ; END
1.2 SET MBBH(2) = VALUE
DEF-FUN INJ:VB
SET INJ:VB = 200

The first immediate command creates the procedure INJ:VB. The second activates it with the variable VALUE set to 200.

Two additional parameters, PAR1 and PAR2, are created when a function defined as

DEF-FUN NAME

is used. In this particular case the calling sequence can have optionally zero, one or two parameters,

e.g. TYPE NAME

or TYPE NAME(2)

or TYPE NAME(2,3)

when PAR1=PAR2=0, or PAR1=2 and PAR2=0 or PAR1=2 and PAR2=3.
Omission of parameters is only allowed in the above case where there is no formal parameter specification, and only for DEF-FUN. If no parameters are wanted in the calling sequence, this can be forced as

\text{DEF-FUN NAME()}

If a second function is defined with the same name as an existing one, the first is not deleted and so two exist with the same name. The first defined is always accessed first. It can be removed using the OPEN command.

11.2 The OPEN Command

This command takes the text from an already defined function and puts it into the working area. The procedure is deleted. It is thus the reverse of the DEFINE command and is used for editing. E.g.

\text{OPEN NAME}

Any previous program or data in the working area is first deleted by this command, just as if it had been proceeded by an "ERASE ALL."

11.3 The VALUE and $VALUE Commands, ERROR Function

The VALUE command ascribes the value of the function when it is used in the read mode. $VALUE does the same for a string function. Both these commands cause the function to terminate and return the specified value. Examples of these commands are

1.9 \text{VALUE X+Z*Y}

1.9 \text{$VALUE "THIS IS THE RESULT STRING"}

If an error is encountered the standard NODAL error mechanism can be activated using the sequence

\text{SET ERROR = X}

where X is the error number. This causes NODAL to enter its error reporting mode as described in section 3.3. A normal return is made from a write function or call function when there are no more lines to be executed, or the END command is encountered.
The Defined Function Area

Normal program lines and data are stored in the "working area" as described in section 8.1. Defined functions are stored in a separate area called the "defined function area". When the DEFINE command is used, the lines of program in the working area are wrapped up with the name of the function and the formal parameter specifications, then transferred to the defined function area. The OPEN command, of course, does the reverse.

When a defined function is in operation its text is in the defined function area, but its variables go into a reduced version of the main program working area. This is created by fore-shortening the working area of the calling program at its level of use when the defined function is called. When the defined function terminates, the working area is given back to the calling program and the local defined function variables are lost. While the defined function is in operation all the normal dynamic properties of the working area are available, so that loading and saving of variables and the use of overlays is allowed. The text, however, is in the defined function area which is not dynamic. Thus attempts to load, erase or change program lines are illegal.

The "ARG's" described in section 8.5 are main program globals. They can be used by defined functions, which access the same set as the main program and overlays.

Utility Commands

The following commands are available for managing defined functions:

LISD
SDEF FILENAME
LDEF FILE1 FILE2 FILE3
ZDEF

LISD causes a list of the current defined functions to be typed out, together with their sizes and parameter specifications.

SDEF causes the contents of the defined function area to be saved on the specified file.

LDEF causes the contents of the specified file(s) to be loaded into the defined function area. Any previous contents are first deleted.
ZDEF clears the DEFINEd function area, and so is the equivalent of "ERASE ALL" for the working area. It is good practice to do a "ZDEF" at the beginning of any program which does not specifically use the defined function area, so as to avoid any clash with variables left over from previous programs.

11.6 Recursion

NODAL defined functions can be used recursively. An example is

```
1.1 IF PAR2>PAR1; VALUE HCF(PAR2,PAR1)
1.2 IF PAR2=0; VALUE PAR1
1.3 VALUE HCF(PAR2,MOD(PAR1,PAR2))
DEF-FUN HCF
```

This example creates a recursive function to find the highest common factor of two numbers. In NODAL the depth of recursion is limited by the working space available. About 70 words are used per level.

In compiler type languages recursive techniques are not usually recommended (where they are possible) on grounds of slow execution. This is not so in NODAL if the recursive program is shorter than the non-recursive, as is often the case.

11.7 Other Uses of the Defined Function Area

The defined function area is a static area associated with the main program working area but which is not affected by normal RUN, OLD, SAVE, etc. commands. In addition to defined functions this area can be used to hold user specific machine code functions, and also named variables and arrays.

Normally the satellite computers have all the machine code data modules required for equipment access built in at system generation. Similarly the console computers have the functions for display and interaction built in. These are sufficient to allow normal control programs to run at a satisfactory speed. Some applications, however, can benefit from complex iterative calculations which reflect the physics of the accelerator. These might be too slow in NODAL. These calculations can then be written as normal machine code NODAL functions and loaded on request into the defined function area using the LDEF command. The functions must exist as NORD BRF modules with the required headers as will be described later.
Any variables and arrays saved on a file by the normal SAVE command can be loaded into the defined function area using the LDEF command. These variables and arrays then form a sort of named common, or context set of variables. They can be used to transmit information between different programs in a particular suite for a given application. As the defined function area is not dynamic, any strings can only be read. An attempt to write will result in an "illegally attempted shuffle". Similarly variables and arrays in the defined function area cannot be the objects of EXECUTE or REMIT commands.

Files of all three types can be loaded with a single LDEF command. Machine code functions should be loaded first so that if subsequent defined functions are opened, no attempt will be made to shuffle a machine code function.
12. Real Time Facilities

Up till now we have only considered interactive programs whose execution is governed by the hardware speed or the requirements from the operator to touch buttons or make other responses. In control applications it is often required that a program be synchronised to "real time". Traditionally this means clock time or an external event. In the SPS there is the added requirement to synchronise with the accelerator cycle time of about 10 seconds.

Two types of synchronisation can be distinguished. One is to synchronise an interactive program with clock time, accelerator cycle time or an event. The other is to schedule the independent execution of a separate program at some clock time, accelerator time, or on the occurrence of an event.

12.1 Synchronising Interactive Programs

Chapter 7 described how a program can be synchronised with the return of data from remotely executed parallel sub-processes using the WAIT command. The WAIT command is also used for time synchronisation in the forms

```
WAIT-TIME TIME
WAIT-CYCLE EVENT,DELAY
```

The first causes the program to wait for a given number of seconds before going on to the next instruction. The commands

```
SET MBBH(1)=456 ; WAIT-TIME 0.5 ; TYPE MBBH(1)
```

cause the computer to set a current, wait half a second, then print out a current.

The second form causes the computer to wait for a given time in the accelerator cycle before continuing. This time is expressed as an event and a time from that event. The latter can be omitted if not required.

A program can be made to wait for the occurrence of an interrupt by the HANG function. For example

```
HANG(10)
```

could cause the program to wait for the occurrence of interrupt number 10. The meaning of interrupt numbers is very installation dependent and so is not elaborated further here.
12.2 Scheduling Independent Programs

Programs can be scheduled for independent execution using the functions

\[
\begin{align*}
SCHEDL & (PROGRAM, ODEV, TIME, INTERVAL) \\
RTRUN & (PROGRAM, ODEV, EVENT, DELAY) \\
REPEAT & (PROGRAM, ODEV, EVENT, DELAY) \\
HOOK & (PROGRAM, ODEV, INTERRUPT)
\end{align*}
\]

SCHEDL causes the program to be started at time TIME and repeated every INTERVAL (not repeated if INTERVAL < 0). ODEV specifies the device on which output can occur. TIME and INTERVAL are specified in seconds. The first parameter PROGRAM can mean different things, depending on the installation. Normally it is the name of the file where the NODAL program is stored. Alternatively it may be the name of a machine code function which is written and executed in the NODAL context but with machine code speed.

RTRUN causes the program to be executed at a given time in the accelerator cycle as specified by EVENT and DELAY.

REPEAT causes the program to be executed at the given time every cycle.

HOOK causes the program to be executed every time the specified interrupt occurs.

\[
\text{DISCON} (PROGRAM)
\]

is a command which disconnects the program from any SCHEDL, REPEAT, or HOOK connection. A program can DISCON itself.

Time is measured in seconds, probably from 1st January at 0.00 hours. A NODAL function "TIME" is available to read out this time. Thus to schedule the program stored on "MY:FIL" for a single execution in ten seconds time one might use

\[
SCHEDL(MY:FIL, 1, TIME+10, 0)
\]

Other functions will be provided in the service computer for conversion of calendar specifications to seconds and vice-versa. The function

\[
\text{LISP}
\]

causes NODAL to print out the state of the programs in the computer, i.e. it tells which programs are HOOKed, SCHEDLed, etc.
12.3 Some Implementation Details

The NODAL interpreter is written as re-entrant code so only one copy is required irrespective of the number of concurrently existing programs. Each concurrently active program, however, requires its own "working area" (sometimes called "task buffer" or "text buffer"). The number of these working areas depends on the system generation, and even on the operating system used. There is usually at least one for interactive programs, one for independent programs, and one to service requests from remote computers.

In computers where there is only one working area for either independent real time programs, or remote executes, it is important that such programs do not wait for significant periods of time. Otherwise the use of the working area will be denied to other users.
13. Writing Machine Code Modules for NODAL

One of the most important features of NODAL and similar interpretive schemes is the ability to add machine code functions written by a user for his specific application. This permits the language to be extended to fit a particular purpose. The application can then be programmed as a series of small modules linked together at run time by the interpreter. The use of these functions was described in chapter 6. In this chapter the essential implementation details are given.

13.1 The Header

When the NODAL interpreter encounters a name in a program, it finds out about the name by searching through several lists. Names can refer to elements of many different types, e.g. variables, arrays, and functions. A summary of all element types and a discussion of the lists is given in the next chapter. Here we are concerned with names that refer to machine code modules. Such a module would be contained in one of the lists with a header as follows:

Data Module Header

<table>
<thead>
<tr>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE = 7</td>
</tr>
<tr>
<td>NAME</td>
</tr>
<tr>
<td>FIRST E.N.</td>
</tr>
<tr>
<td>LAST E.N.</td>
</tr>
<tr>
<td>CORE ADDRESS</td>
</tr>
<tr>
<td>MASS ADDRESS</td>
</tr>
<tr>
<td>CODE</td>
</tr>
</tbody>
</table>
General Purpose Function Header

<table>
<thead>
<tr>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE = 8 TO 12</td>
</tr>
<tr>
<td>NAME</td>
</tr>
<tr>
<td>8 7 6 5</td>
</tr>
<tr>
<td>4 3 2 1</td>
</tr>
<tr>
<td>CORE ADDRESS</td>
</tr>
<tr>
<td>MASS ADDRESS</td>
</tr>
<tr>
<td>CODE</td>
</tr>
</tbody>
</table>

Parameter Descriptors

The first word in the header gives the size of the element which is the connection to the next element in the list. The size must be positive and the list is terminated by -1.

The second word gives the type of the element. In this chapter we are concerned only with type 7 which denotes data modules and equipment functions, and with types 8 to 12 which denote general purpose machine code functions. The other element types between 1 and 25, which include some other types of machine code module, are summarised in the next chapter.

Then we have three words which give the name of the module. These are followed by two words which indicate how the module should be called. For general purpose functions these two words give the number and type of the parameters. These will be discussed in more detail in the next section.

For data modules the two parameter descriptor words give the first and last equipment numbers of the given name in the data module. Thus several headers can be used for the same data module with different names for different ranges of equipment numbers. NODAL adds the unit number of that name, as specified in the calling program, to the first equipment number of that name, as specified in the header, and subtracts one to get the correct equipment number to send to the data module. This is of course checked against the maximum for that name as given in the header. Normally one header is provided which covers all equipments in the data module sequentially from the first to the last existing. The name in this header is called the "generic" name of the data module.
If the second of the two parameter descriptor words for a data module is -1 then an equipment function call is required in the NODAL program and no unit number is given. The equipment number passed to the data module is that contained in the first word. It is interesting to distinguish this from the case where the first and last equipment numbers are the same. In this case a data module call is indicated, but the unit number can only be 1 and again it is the unit number in the first word which is transferred to the data module. The equipment function option is useful to designate one and only one equipment by that name so as to make a unit number specification unnecessary. The second case might be used where only one of that name exists, but the possibility of adding another in the future is foreseen. In any case this makes no difference to the actual data module itself.

Finally in the header we have two words giving the entry address of the module. The first gives the address in the 16 bit program space. The second gives the mass storage address which is the core load number in SYNTRON and the segment number in SINTRAN III. This latter word is usually set to zero by the programmer in the header included with the body of the module. Other lists can be made up at run time, however, in which this word is not zero.

13.2 Parameters and Data Modules

For NODAL elements types 7 to 12, parameter passing is done in the NORD standard FORTRAN way, the A-register pointing to a list of parameter addresses.

Data modules always have four parameters. These are

a. VALUE. This is the data to be written to the equipment, or returned after reading. In normal read-write accesses it is a three word floating point quantity. In the array handling mode it is an array. The address points to the beginning of the array data, and it is up to the data module to check the size and type of the array from its preceeding descriptor (see next chapter for details of the array element).

b. FLAG (integer). This indicates whether the operation is a read or a write, simple or array. +1 means simple read, -1 simple write, +2 array read, -2 array write. On return FLAG must be set to zero if all is OK, otherwise to an error number.
c. EQUIPMENT NUMBER (integer). This indicates the particular equipment in question.

d. PROPERTY (integer). This indicates which aspect of the equipment is to be read or written.

The data modules will normally check the parameters for legality. In many cases they will also want to check the section or capability of the user. These are not passed as explicit parameters, but can be obtained by a call to the routine "PASWD".

13.3 Parameters for General Purpose Functions

The number of parameters in the machine code calling sequence for general purpose functions depends on both the type of function and the parameter descriptor words. The parameter descriptor words control the number and type of the "explicit" parameters, those which appear in the NODAL calling sequence. Preceding these, however, NODAL supplies 2, 1, or 0 "implicit" parameters, depending on the type of the module. Types 8, 9, and 10 are respectively read-write, write only, and read only functions. They have two implicit parameters. The first is the three word floating point value which is to be read or written. The second is the flag (integer) which is +1 for read and -1 for write at entry, and which returns the error number (0 for OK) at exit, just as for data modules above. Type 11 is a "CALL" function with one implicit parameter, the flag. This is set to zero at entry and returns the error code on completion as described above. Type 12 are call functions with no implicit parameters, so the actual parameters correspond exactly with those in the NODAL calling sequence. Such functions can never "fail" in the NODAL sense. Any errors must be handled by the NODAL program via the explicit parameters.

Up to eight explicit parameters can be specified by the two parameter descriptor words in the header. The parameter descriptors are four bit fields arranged as shown in the figure. A zero field denotes the end of the descriptors. Thus fifteen different kinds of parameters can be specified, only nine being implemented at present. These are given in the table below, the number being the value required in the four bit field to specify the particular type of parameter.

PARAMETER TYPES

1  real value - 3 word floating point number
2 integer value - 1 word integer
3 string value - 4 word string descriptor
4 NODAL reference - NODAL data element, address of first word, i.e. size
5 real reference - 3 word floating reference
6 integer reference - 1 word integer reference
7 real array - 3 word per element floating array
8 integer array - 1 word per element integer array
9 NODAL name - 3 word element

In the case of real and integer value parameters an expression can be used to define the parameter, e.g.

SET CAMAC(SIN(PIE/2),N,0,16) = 5

could give four integer value parameters.

All others parameters must be simple names, e.g.

CALL FFT(A,-1)

where the first parameter is an array name.

Logically array elements should be usable in response to specifications 5 and 6 but this is not implemented at present. Type 6 is used when the subroutine wishes to return an integer to the calling program. NODAL only deals in real simple variables, however, so automatic conversion is performed at entry and exit from the routine. This conversion will malfunction at present if the same name is used in response to more than one type 6 specification in the same routine.

13.4 String Descriptors

Strings require further explanation. These are widely used both in string functions and internally in NODAL. A string is described by a four word descriptor as follows:

word1 - start address of string area, word address
word2 - reader pointer. First unread byte in string.
word3 - writer pointer. First unwritten byte in string.
word4 - end pointer. Maximum number of bytes possible.
Word 1 contains the word address of the start of the string area. The first byte position is in the left hand 8 bits. The other three words are byte displacements. Normally a string is considered to exist between the reader and writer pointers. NODAL has a number of useful internal routines to manipulate strings which can be used within a function. These routines are normally accessed with the X-register containing the address of the string descriptor.

13.5 Working Space

NODAL provides a stack structure so that the writing of re-entrant and recursive modules is normal. All functions consist only of code and constants. The local storage is allocated on the stack by NODAL for recursive calls. The operating system provides a different stack for each NODAL activation.

The stack is based on the B-register. Only positive displacements are used for local variables. The first 16 negative displacements are always available to the module but will be overwritten by subsequent subroutine calls. Their use is reserved for some high speed service routines. Any amount of local variable storage positive to the B-register can be declared, but of course only 127 locations can be directly addressed. This is normally more than sufficient.

The first 9 local variables are used by the NODAL run time system as follows:

0 =PREV     - pointer to previous stack position
1 =TREG     - storage for T-register
2 =AREG     - storage for A-register
3 =DREG     - storage for D-register
4 =XREG     - storage for X-register
5 =LREG     - storage for L-register
6 =GLOBL    - pointer to global area
7 =STK      - stack routine address
8 =USTK     - unstack routine address

The T, A, D, X, and L-register locations contain copies of the registers at entry to the routine. At exit the registers are latched from these locations.
13.6 Support Macros

In order to help the programmer, improve the appearance and readability of programs, and to reduce the chance of error, NODAL provides a number of macros to support its subroutine structure. These control the passing of parameters, the declaration of local variables (naming and reservation of space), subroutine entry and exit (with stacking and unstacking), and filling in the size of the header.

An example of a simple function using some of these macros is shown in appendix 3. Line 5 shows the header. The first word, the size of the module, is left at zero. The correct size is filled in by the macro 8FINN at the end in line 47. This also deletes all symbols defined in the module except the entry point as specified in the 8FINN macro.

The macros PROGM, FPAR and DATA on lines 7 to 16 do not generate code but set up the parameters for the main entry macro USENT on line 18. PROGM declares the start of a module. FPAR declares a local variable into which a parameter address will be loaded. DATA declares a named local block of storage, the parameter after the comma giving the number of words required. The FPAR declarations must be in the right order and must precede the DATA declarations.

The macro USENT uses the information gleaned by PROGM, FPAR and DATA to do three things. First it adjusts the B-register to give the required local storage. Secondly it transfers the parameter addresses into the locations specified by FPAR. Thirdly it sets the registers to the calling values after storing them in the local variables as specified in section 13.5 above.

Then follows the specific code for the function. On line 37 we have the macro USRET which achieves the return from the function, resetting the B-register to its previous value. Return is not made through the actual value of the L-register but through the local variable LREG in which was stored the original value of the L-register.

Note that parameters are easily accessed using indirect B-register addressing, and local variables use direct B-register addressing. This routine is a type 10 read-write function and so has the two implicit parameters VALUE and FLAG in addition to the four explicit integer value parameters specified in the header on line 5.
Three other macros exist but are not used in this example. These are ENTER, RET, and 8COMN. ENTER and RET perform the same functions as USENT and USRET in the case where there is no standard parameter list pointed to by the A-register. FPAR is meaningless in a module starting with ENTER. Any parameters must be passed using the T, A, D, and X registers. Although USENT with no FPAR's would do the same job, ENTER is shorter (4 words instead of 6) and faster. ENTER and RET are widely used in functions other than types 7 to 12, and also form the basis of the internal NODAL implementation. The macro 8COMN is used to extend the size of a header to encompass a subsequent routine with no header but whose entry point is to be declared. The sequence must be the 8FINN as before, followed by 8COMN which takes no parameter, followed by the coding of the support routine, followed by another 8FINN with the entry address of the support routine as its parameter. The support routine can be written using either USENT or ENTER depending on the parameter passing required.

13.7 Other Types of Module

As mentioned previously NODAL provides a wealth of different module types as the ability to add modules is considered very important. FORTRAN library functions, free functions, string functions, and pattern functions are somewhat more specialized so examples are not given here. If a new one is required, the best approach is to select an existing function, say from the list in appendix 5, as an example and get a copy of the source code.

Data modules are of course very general and important but no example is given. This is because a wealth of instructions and support facilities have been provided which are outside the scope of this manual. The person in charge of the data modules for a given installation should be contacted for examples and instructions.
14. NODAL Element Lists

Much of the flexibility of NODAL comes from the fact that lines of program and named data are packaged into self-contained "elements" which are searched for at run time. Many of these elements are completely relocatable and computer independent. This feature provides the basis of NODAL's multi-computer capability. Machine code module elements, however, are anchored to their particular computer, either at system generation or after relocatable loading.

These elements can be contained in several lists. The most important is in the "working area" described in chapter 8. This list is completely dynamic and can contain only relocatable elements, namely lines of program, variables, and arrays. Defined functions can also be loaded into the working area but cannot then be used recursively. The contents of this list can be examined using the LIST command and the LISV function.

The defined function area is less dynamic but can be reconfigured using the LDEF command as described in chapter 11. This list can contain defined functions, variables and arrays, and machine code modules which are relocatably loaded. The LDEF command contains a loader which can load relocatable modules and satisfy a certain number of references into the interpreter. These references can be listed using the LISE function. The contents of the defined function area are listed using the LISD function.

The other lists are set up at system generation. These lists are global to all programs and usually contain functions and data modules. They can, however, contain variables and arrays for special purposes. Normally one list, printed out by LISR, is contained in the mass storage address space of the interpreter itself, and another, printed out by LUST, exits in multiple versions on different mass storage core loads or segments. This latter can exist in as many versions as are necessary to hold all the data modules.

All NODAL elements have as their first word the size of the element, which is used to link to the next element. The second word is always the type of the element. Except for type 1 which is a line element, all others are named elements, and the name is stored in the next three words.
14.1 Element Types

A summary of element types is given in appendix 4. This section gives more information on each type.

Type 1 - Line Element

The structure of a line element is shown below:

| SIZE | TYPE | LINE NUMBER | NO. OF BYTES | TEXT |

This element starts with the size and type then has a word which contains the line number, stored with group number in the left hand byte and the step number in the right hand byte. Then follows a fourth word which gives the number of bytes of text in the line. These four header words are then followed by the text of the line, less of course the line number.

Type 2 - read/write variable

This is an eight word entry comprising the usual size, type and name, plus the three words giving the floating point value of the variable.

Type 3 - read only variable

This is the same as type 2 but NODAL cannot change the value, only read it. An example is PIE, which is as follows (octal numbers)

10; 3; #PI; #E$-##$; 0; 040002; 144417; 155242

Type 4 - read/write numerical array

In addition to the header an array element has two descriptor words. The first gives the number of rows and the type. The type occupies the rightmost four bits, i.e. bits 0-3. It contains 1 for integer, 3 for floating. The leftmost 12 bits give the number of rows so the maximum number of rows is 4095. The second descriptor word contains the total number of elements which is of course an integer multiple of the number
of rows. Thus a 64 word integer array could be as follows

107; 4; #AR; #RA; #Y$-$#$; 2001; 100; *+100/

Type 5 - Read Only Numerical Array

This is exactly as type 4 but can only be read. It is useful for incorporation in built in lists for fixed data. The read only property can be ignored by machine code modules, so such arrays can be used to communicate data from a machine code process to all NODALS.

Type 6 - FORTRAN Library Function

This entry permits NODAL to call a FORTRAN library function. Two words are used in addition to the header, the number of parameters and the entry point of the routine. The number of parameters can be 1 or 2. The first parameter is held in the TAD and the second pointed to by X. The result is returned in TAD, X being destroyed. The routine does a skip return if OK otherwise direct return with A = error number. The routine can use locations B-20 to B-1 as scratch. The NODAL library functions listed in section 6.2. are called using such headers but can also be used in user functions according to the above rules.

Types 7 to 12 - Machine code modules

These are described in chapter 13.

Types 13, 14 and 15 - NODAL defined functions

These are created by the DEFINE command and will not normally be used in a user list. The structure is as shown below
The number of parameters can be -1, 0 or 1-8. -1 is obtained by the DEF-FUN command with no arguments and causes the variables PAR1 and PAR2 to be created as described in chapter 8. 0 means no parameters, and 1 to 8 gives the number of parameters required. For each parameter a descriptor block of 4 words is used. The first gives the parameter type, 1 for value, 2 for reference, 3 for string. The remaining 3 give the formal parameter name.

Types 16 and 17 - String variables and arrays

These element types are created by the $SET and DIM-STR commands. As their size is dynamic they can not be used in user lists. Their structure is as follows:
Type 18 - Free machine code routine

This element type allows the user full freedom in defining the syntax of the parameter list. He must, however, use the internal routines of NODAL to get working space, process parameters, etc. The header consists simply of the size, type, name, plus the entry point of the routine. Information is passed to the routine as follows:
D = FLAG - 0 for call, +1 for read, -1 for write

X = POINTER to string for arguments
    String starts at first character after routine name, must be completely read by routine.

A = POINTER to value for write function

T = POINTER to element header

TAD = VALUE return from read function

The routine must do a skip return if OK, or direct return with
A = ERROR NUMBER.

Type 19 - Pattern variable

The structure of a pattern variable is as follows:

```
+-------------------+
| SIZE              |
| TYPE = 19         |
|                   |
|                   |
| NAME              |
|                   |
|                   |
| ALTERNATIVE       |
| PATTERN           |
```
The structure of each alternative pattern is as follows:

```
SIZE
NO. OF PRIMARIES
CONTROL WORD

SIZE
TYPE
CONTROL WORD
SIZE OF ELEMENT

[PRIMARY ELEMENT]

[ASSIGNMENT]
```

Type 21 - Pattern function

The header for a pattern function is the same as that for a free function type 18, i.e. size, type, name and entry point of the function. Again NODAL facilities must be used for working space and parameters. The register use is as follows:

- \( T \) = Argument string
- \( X \) = Subject string
- Return AD = reader and writer pointers for match
- Failure return 1 for fail
- Failure return 2 for abort
Types 22, 23 and 24 - String functions

These are similar to type 18 free functions but with the register usage as follows:

D = FLAG for read/write functions
   +1 for read, -1 for write

X = Pointer to argument string

A = Pointer to result string for read,
   string to be output for write

T = Pointer to element header

Skip return for normal return, failure return A = ERROR NUMBER, special error number SERR2 indicating failure jump requested for $SET command.

Type 25 - Reference pointer

Created temporarily at run-time in a defined function to point to reference variable.
(CTRL)A  Delete one character (echoes ^)
(CTRL)C  Copy one character from old line to new line
(CTRL)S  Skip one character of old line
(CTRL)E  Start or stop inserting new characters, echoes < or >
(CTRL)H  Copy up to end of old line
(CTRL)Y  Append rest of old line to new line and edit result
(CTRL)M  Or RETURN, terminate edit
(CTRL)D  Copy up to end of old line and terminate edit

(CTRL)OC  Copy old line up to but not including character C
(CTRL)ZC  Copy old line up to and including character C
(CTRL)PC  Skip old line up to but not including character C
(CTRL)XC  Skip old line up to and including character C

(CTRL)W  Delete last word, echoes \n(CTRL)Q  Delete back to beginning of new and old lines
(CTRL)VC  Take character C literally, even if control character
(CTRL)R  Retype fast
(CTRL)T  Retype aligned
(CTRL)L  Terminate edit, same as (CTRL)M, or RETURN
(CTRL)F  Copy rest of old line, as (CTRL)D, but without typing
APPENDIX 2
ERROR DEFINITIONS

Error Numbers

NODAL error numbers can be in the range 1 to 127. They may vary between installations, but the standard arrangement is that the numbers 1 to 63 correspond to individual "canned" messages, and that numbers 64 to 127 are printed out with a numeric code for system specific allocation.

The plain language printout for any error number can be obtained on-line using the ERMS function. The result of the command

FOR I=1,63; TYPE I ERMS(I)!

is given below.

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
APPENDIX 2
ERROR DEFINITIONS

38  NOT IMPLEMENTED
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  DEFINED FUNCTION AREA FULL
53  SYNTAX ERROR IN DEFINE COMMAND
54  ILLEGAL STRING 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
APPENDIX 3
EXAMPLE OF MACHINE CODE MODULE

1  %% SIMPLE NON-INTERRUPT CAMAC ROUTINE FOR NORD-10 NODAL
2  %% SET CAMAC(C,N,A,F)=Z -- SET Z=CAMAC(C,N,A,F)
3  %% CAMAC STATUS REGISTER RETURNED FOR CONTROL FUNCTIONS
4  
5  0; 10; #CA; #MA; #CS-##$; 0; 21042; CAM1; 0
6  
7  PROGM
8  CAM1=*"FPAR VALUE
9  FPAR FLAG
10  FPAR CRATE
11  FPAR MODUL
12  FPAR SUBAD
13  FPAR FUNCT
14  DATA CRIO,1
15  DATA NAF,1
16  
17  USENT
18  LDA I CRATE,B; JAN CAMER; AAA -20; JAP CAMER
19  AAA 20; SHA 6; STA CRIO,B
20  LDA I MODUL,B; JAN CAMER; AAA -40; JAP CAMER
21  AAA 40; SHA 11; STA NAF,B
22  LDA I SUBAD,B; JAN CAMER; AAA -20; JAP CAMER
23  AAA 20; SHA 5; ORA NAF,B; STA NAF,B
24  LDA I FUNCT,B; JAN CAMER; AAA -40; JAP CAMER
25  AAA 40; ORA NAF,B; STA NAF,B
26  LDA I FUNCT,B
27  AAA -10; JAN CAMR
28  AAA -10; JAN CAMC
29  AAA -10; JAN CAMW
30  CAMC, LDA I FLAG,B; JAN CAMER
31  LDX CRIO,B; LDT (IOX 2037; RORA DT SX
32  LDA NAF,B; EXR ST
33  LDT (IOX 2070; RORA DT SX; EXR ST
34  CAM2 NLZ 20; STF I VALUE,B
35  CAM2A, SAA 0
36  CAM3, STA I FLAG,B; USRET
37  CAMER, SAA 24; JMP CAM3
38  CAMR, LDA I FLAG,B; JAN CAMER
39  LDA (IOX 2037; LDT CRIO,B; RORA DT SA
40  LDA NAF,B; EXR ST; JMP CAM2
41  CAMW, LDA I FLAG,B; JAP CAMER; LDX CRIO,B
42  LDF I VALUE,B; JPL I (NFX
43  LDT (IOX 2021; RORA DT SX; EXR ST
44  LDA NAF,B; LDT (IOX 2037; RORA DT SX
45  EXR ST; JMP CAM2A
46  8FINN CAM1
<table>
<thead>
<tr>
<th>Element Type</th>
<th>Internal Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. LINE ELEMENT</td>
<td>DIRECT</td>
</tr>
<tr>
<td>2. SIMPLE VARIABLE</td>
<td>DIRECT</td>
</tr>
<tr>
<td>3. READ ONLY VARIABLE</td>
<td>DIRECT</td>
</tr>
<tr>
<td>4. NUMERIC ARRAY</td>
<td>ARRAY</td>
</tr>
<tr>
<td>5. READ ONLY NUMERIC ARRAY</td>
<td>ARRAY</td>
</tr>
<tr>
<td>6. FORTRAN LIBRARY FUNCTION</td>
<td>GFUN</td>
</tr>
<tr>
<td>7. DATA MODULE AND EQUIPMENT FUNCTION</td>
<td>SYSVR</td>
</tr>
<tr>
<td>8. READ/WRITE GENERAL PURPOSE MODULE</td>
<td>ASSFN</td>
</tr>
<tr>
<td>9. WRITE ONLY GENERAL PURPOSE MODULE</td>
<td>ASSFN</td>
</tr>
<tr>
<td>10. READ ONLY GENERAL PURPOSE MODULE</td>
<td>ASSFN</td>
</tr>
<tr>
<td>11. CALL, NODAL ERROR RET G.P. MODULE</td>
<td>ASSFN</td>
</tr>
<tr>
<td>12. CALL NO ERROR RET G.P. MODULE</td>
<td>ASSFN</td>
</tr>
<tr>
<td>13. READ/WRITE DEFINED FUNCTION</td>
<td>NODFN</td>
</tr>
<tr>
<td>14. CALL DEFINED FUNCTION</td>
<td>NODFN</td>
</tr>
<tr>
<td>15. NODAL READ/WRITE STRING FUNCTION</td>
<td>NODFN</td>
</tr>
<tr>
<td>16. STRING VARIABLE</td>
<td>DIRECT</td>
</tr>
<tr>
<td>17. STRING ARRAY</td>
<td>DIRECT</td>
</tr>
<tr>
<td>18. FREE MACHINE CODE ROUTINE</td>
<td>DIRECT</td>
</tr>
<tr>
<td>19. PATTERN VARIABLE</td>
<td>DIRECT</td>
</tr>
<tr>
<td>20.</td>
<td></td>
</tr>
<tr>
<td>21. PATTERN FUNCTION</td>
<td>DIRECT</td>
</tr>
<tr>
<td>22. READ/WRITE STRING FUNCTION</td>
<td>DIRECT</td>
</tr>
<tr>
<td>23. WRITE ONLY STRING FUNCTION</td>
<td>DIRECT</td>
</tr>
<tr>
<td>24. READ ONLY STRING FUNCTION</td>
<td>DIRECT</td>
</tr>
<tr>
<td>25. REFERENCE POINTER</td>
<td>GNAM</td>
</tr>
</tbody>
</table>
Function Documentation

This appendix gives a description of some of the most common functions. The documentation is extracted by a computer program from the source listing of the function code. Normally this sort of documentation is supplied in a much more complete form as a separate manual which will cover all functions and data modules for a given installation. For a particular computer the available functions and data modules can be found from a "LISR" printout, then the manual consulted for further details.

The NODAL program which was used to extract the documentation from the source files is listed at the end of the function documentation.

IDEV    SET OR READ INPUT COMMAND DEVICE ASSIGNMENT
        SET Z=IDEV, SET IDEV=Z

ODEV    SET OR READ COMMAND OUTPUT DEVICE ASSIGNMENT
        SET Z=ODEV, SET ODEV=Z

TIME    GET INTERNAL TIME IN SECONDS
        NODAL TYPE 10 READ ONLY FUNCTION

DATE    STRING FUNCTION RETURNING DATE
        CURRENT DATETIME IN ISO FORMAT
        MM-DD-HH:MM:SS

FCOPY   COPY A FILE FROM SOURCE TO DESTINATION
        ANY ARBITRARY FILE FORMAT
        CALLED AS - FCOPY("SOURCE","DESTINATION")
LISTV LIST PROGRAM VARIABLES

FOLLOWING FORMAT FOR OUTPUT
COL 1 VARIABLE NAME
COL 2 SIZE OF NODAL ELEMENT
COL 3 TYPE
COL 4 CURRENT VALUE OF VARIABLE

LISR LIST USER FUNCTIONS

FOR OUTPUT FORMAT SEE "LUST"

LISA LIST INBUILT FUNCTIONS

FOR OUTPUT FORMAT SEE "LUST".

LUST LIST USER FUNCTIONS

LUST(N) WHERE N = CORELOAD NUMBER
FOLLOWING FORMAT FOR OUTPUT
COL 1 CORE ADDRESS (OCTAL) OF FUNCTION
COL 2 NAME OF FUNCTION
COL 3 SIZE (DECIMAL) OF FUNCTION
COL 4 FUNCTION TYPE AND CALLING SEQUENCE
LIST TERMINATES WITH OCTAL ADDRESS
OF END OF ELEMENT LIST
THE TOTAL NUMBER OF FUNCTIONS, AND THE TOTAL NUMBER
OF WORDS USED.

HELP LIST NODAL COMMANDS

GIVES NODAL COMMANDS AND THEIR MINIMUM ABBREVIATION.
FOUR COMMANDS PER LINE.

LISD LIST DEFINED FUNCTIONS

ALSO GIVES SIZE OF DEFINED FUNCTION AREA

LISDM LIST DATA MODULE INFORMATION

CALL LISDM(SYSTEM VARIABLE NAME)
LISTS COMPUTERS IN SYSTEM

GIVES THE STANDARD MNEMONICS FOR THE COMPUTERS IN THE SYSTEM AND THEIR CORRESPONDING NUMBERS.

BIT FUNCTION

SET BIT(N,A)=Z; SET Z=BIT(N,A)
A=DESTINATION, CAN BE ANY SETTABLE ITEM
N=BIT NUMBER, 0 TO 15
SET OR READ A SPECIFIC BIT IN A SUPPOSED 16 BIT WORD

DO LOGICAL INCLUSIVE OR

SET Z=IOR(M,N)
INCLUSIVE OR OF (M,N) INTO Z
RETURNS INCLUSIVE 'OR' OF TWO INTEGER ARGUMENTS

DO LOGICAL AND

SET Z=AND(M,N)
LOGICAL PRODUCT OF (M,N) INTO Z
RETURNS AND OF TWO INTEGER ARGUMENTS

DO LOGICAL COMPLEMENT

SET Z=NEG(M)
INVERTS ALL BITS; RETURNS COMPLEMENT OF INTEGER ARGUMENT

DO LOGICAL SHIFT

SET Z=SHIFT(EXPR,N)
EXPR IS ANY EXPRESSION EVALUATED AND ROUNDED TO INTEGER
N IS NUMBER OF SHIFTS, +VE FOR SHIFT LEFT, -VE FOR RIGHT
LOGICAL SHIFT, IE. ZEROS PUT INTO VACATED BITS

RETURN SIZE OF ARRAY

GIVES NUMBER OF ELEMENTS IN ARRAY
ONE PARAMETER, NAME OF ARRAY
ERROR MANIPULATE ERROR SITUATION
SET ERROR=EXPRESSION, PROVOKES NODAL ERROR
POSITIVE ERROR NUMBER MUST BE GIVEN
OTHERWISE ERR37 VALUE OUT OF RANGE IS GIVEN
SET Z=ERROR, RETURNS ERROR SET ON DO ERROR JUMP

LOC SET OR READ MEMORY LOCATION
SET LOC(I)=[XXXXXXXX
WARNING! THIS FUNCTION CAN DAMAGE THE CONTROL SYSTEM
IT IS FOR THE USE OF SPECIALISTS ONLY

ERMES RETURN ERROR MESSAGE
ERMES(N) WHERE N = ERROR NUMBER

STRARG STRING ARGUMENT IN GLOBAL AREA
USES 21 DECIMAL, 25 OCTAL GLOBALS FROM 9STAG
STAG CONTAINS WRITER POINTER
STAG+1 TO 9STAG+24 CONTAIN STRING, SPACE FOR 40 CHARACTERS
READ-WRITE STRING FUNCTION

NODLIN GET OR CREATE NODAL LINES
SET NODLIN(LN)=STRING, CREATES LINE OF NUMBER LN
SET S=NODLIN(LN), RETURNS LINE WITHOUT NUMBER
READ-WRITE STRING FUNCTION
STRING FUNCTION FAILURE IF NO SUCH LINE

CHAIN RUN CHAINED FILES
CALL CHAIN FN [FN] [FN].....
RUNS CONSECUTIVE FILES RETAINING DATA IN TEXT BUFFER
ANY NUMBER OF FILES MAY BE RUN - UPTO END OF COMMAND.

ALPHA PRE-DEFINED STRING VARIABLE
RETURNS A TO Z
NUM  PRE-DEFINED STRING VARIABLE
     RETURNS 0 TO 9

ASCII GET ASCII VALUE
     VALUE=ASCII(CONC)
     FORMS ASCII VALUE OF STRING BY ADDITION OF ALL CHARACTERS

SIZE FIND SIZE OF CONCATENATION
     VALUE=SIZE(CONC)
     RETURNS NUMBER OF CHARACTERS IN A CONCATENATION

SUBS SUBSTRING
     STRING = SUBS-BEGINNING,END,CONC)
     OR $SET SUBS-BEGINNING,END,STRING) = STRING
     RETURN SPECIFIED SUBSTRING OF GIVEN CONCATENATION

EVAL EVALUATE STRING EXPRESSION
     VALUE=EVAL(CONC)
     TREATS CONC AS AN ARITHMETIC EXPRESSION AND
     RETURNS ITS VALUE

FIND FIND INDEX
     VALUE=FIND-ARRAY-NAME,CONC)
     RETURNS INDEX OF CONC IN ARRAY
     RETURNS -1 IF NOT FOUND

SORT SORT STRING ARRAY
     SORT(ARRAY-NAME, + OR -)
     SORT STRING IN ASCENDING OR DESCENDING ORDER
RUNOFF TEXT PROCESSOR FUNCTION.

\$SET S=RUNOFF(CONCATENATION)
THE ALPHABETICS IN THE CONCATENATION ARE CONVERTED
TO LOWER CASE. HOWEVER, THE CONVERSION OBEYS
CERTAIN SPECIAL CONTROL CHARACTERS:
(/SLASH) THE NEXT CHARACTER IS CONVERTED
TO UPPER CASE.
// (TWO SLASHES) THE SUCCEEDING CHARACTERS ARE ALL
CONVERTED TO UPPER CASE.
\// (2 BACK SLASHES) THE SUCCEEDING CHARACTERS ARE ALL
CONVERTED TO LOWER CASE.
# (HASH) ONE SPACE
_ (BACK ARROW) INTERPRET THE NEXT CHARACTER
LITERALLY.
ALL OTHER CHARACTERS ARE COPIED UNMODIFIED EXCEPT
THAT PARITY IS REMOVED.

ARB MATCH ANY CHAR OR CHAR STRING

POS MATCH A SPECIFIC LOCATION IN STRING
POS(I) I STARTS WITH 0 AT BEGINNING OF STRING

RPOS REVERSE POSITION MATCH
RPOS(I) SIMILAR TO POS(I), BUT POSITION I IS CountED FROM
END OF STRING
RPOS(0) IS JUST TO THE RIGHT OF THE LAST CHARACTER

LEN LENGTH MATCH
LEN(I) MATCHES ANY STRING OF LENGTH (I)

TAB
TAB(I) MATCH UP TO POSITION I
RTAB  REVERSE TAB

RTAB(I)
SIMILAR TO TAB, BUT MATCHES UP TO THE
I-TH POSITION FROM THE END OF THE STRING

SPAN

SPAN(CONC)
MATCH THE LONGEST RUN OF CHARACTERS CONTAINED IN
CONC, STARTING FROM THE CURRENT CURSOR POSITION

BREAK

BREAK(CONC)
MATCH UPTO BUT NOT INCLUDING ANY BREAK CHARACTER
CONTAINED IN CONC

ANY

ANY(CONC)
MATCH ANY SINGLE CHAR CONTAINED IN CONC

NOTANY

NOTANY(CONC)
MATCH ANY SINGLE CHARACTER NOT CONTAINED IN CONC

FAIL  FAIL PATTERN

IF TRIED CAUSES PATTERN MISMATCH AND SO CAUSES OTHER
ALTERNATIVES TO BE TRIED

ABORT  ABORT PATTERN

IF TRIED CAUSES THE COMPLETE FAILURE OF THE $MATCH
COMMAND I.E. NO OTHER ALTERNATIVES ARE TRIED
INBT  NODAL FUNCTION TO INPUT A BYTE FROM A FILE

VALUE=INBT(F) WHERE F = FILE LOGICAL NUMBER
RETURNS FULL 8 BIT VALUE OF BYTE AS POSITIVE INTEGER

OUTBT  OUTPUT A BYTE TO A FILE

SET OUTBT(F)=Z
EXPRESSION Z IS FIXED TO INTEGER,
THEN LOWER 8 BITS OUTPUT TO FILE
F = FILE LOGICAL NUMBER

OPEN  VALUE = OPEN('R' OR 'W' OR 'A',STRING)

OPEN A FILE GIVEN BY THE STRING
FOR READ, WRITE OR APPEND
RETURN FILE NUMBER

CLOSE  CLOSE(FILE NUMBER OR -1)

-1 CLOSES ALL OPEN FILES

OUTPUT $SET OUTPUT(FILE NUMBER)=CONCATENATION

OUTPUT A STRING FOLLOWED BY CRLF
X = POINTER TO ARGUMENT STRING
A = STRING TO BE OUTPUT
RETURN
FAILURE RETURN A = ERROR NUMBER

OUTC  $SET OUTC(FILE NUMBER)=CONCATENATION

OUTPUT A STRING TO A FILE
X = POINTER TO ARGUMENT STRING
A = STRING TO BE OUTPUT
RETURN T = FILE NUMBER
FAILURE RETURN A = ERROR CODE
APPENDIX 5
FUNCTION DOCUMENTATION

INPUT  STRING = INPUT ((FILE NUMBER))

  INPUT A LINE FROM A FILE
  X = POINTER TO ARGUMENT STRING
  A = POINTER TO RESULT STRING
  RETURN
  FAILURE RETURN A = ERROR CODE
  A = SERR2 IF EOF

INPC  STRING = INPC ((FILE NUMBER))

  INPUT A CHARACTER FROM A FILE
  X = POINTER TO ARGUMENT STRING
  A = POINTER TO RESULT STRING
  RETURN
  FAILURE RETURN A = ERROR CODE
  A = SERR2 IF EOF

RWDATA READ OR WRITE FORTRAN FORMAT DATA

  CALLED AS  RWDATA(CONC,FILE!DEV,LIST)
  CONC IS RF!RL!WF!WL!AF!AL FOR READ, WRITE, OR APPEND
  ON FILE (FILE OPENED FIRST, CLOSED AFTERWARDS),
  OR ON LOGICAL NUMBER
  SECOND PARAMETER IS CONC FOR FILENAME,
  OR EXPRESSION FOR LOGICAL NUMBER
  LAST PARAMETER IS A LIST OF VARIABLES OR ARRAYS
  NODAL FREE FUNCTION
APPENDIX 5
FUNCTION DOCUMENTATION

1.10 TYPE "OUTPUT FILE TO BE SPECIFIED FIRST"
1.20 $ASK "OUTPUT FILE" OF; SET N.OF=OPEN("W",OF)
1.40 IF SIZE(IF)=0; CLOSE(N.OF); END
1.50 SET N.IF=OPEN("R",IF)

2.10 DIM-S TEXT
2.20 $SET S=INPUT(N.IF):2.9
2.30 $MATCH S POS(0) "%%" NOTANY("%"):2.2
2.40 $SET TEXT(ARSIZE(TEXT)+1)=S
2.50 $SET S=INPUT(N.IF):2.8
2.60 $MATCH S POS(0) "%%" NOTANY("%"):2.7; GOTO 2.4
2.70 DO 10; GOTO 2.1
2.80 DO 10
2.90 CLOSE(N.IF); GOTO 1.3

10.01 % OUTPUT CONTENTS OF ARRAY TEXT TO FILE IN REQUIRED FORMAT
10.10 FOR I=1,ARSIZE(TEXT); $MATCH TEXT(I) POS(0) BREAK(ALPHA)="
10.20 $MATCH TEXT(1) POS(0) SPAN(ALPHA) $ NM = ""
10.30 $M TEXT(I) POS(0) BREAK(ALPHA) = ""
10.40 $SET OUTPUT(N.OF)=".BLANK 3"
10.50 $SET OUTPUT(N.OF)=".TEST-PAGE " %2 ARSIZE(TEXT)+1
10.60 $SET OUTPUT(N.OF)=NM &7-SIZE(NM) TEXT(1)
10.65 $SET OUTPUT(N.OF)=" "
10.70 IF ARSIZE(TEXT)<2; RET
10.80 FOR I=2,ARSIZE(TEXT); $SET OUTPUT(N.OF)=&7 TEXT(I)
**Command Summary**

NODAL commands can be abbreviated provided there is no ambiguity. Commands, whether abbreviated or not, must be followed by a space or some non-alphabetic character. The first three letters of a command will always be unambiguous, though sometimes fewer will be required. The function HELP can be used to list the NODAL commands available, together with the minimum abbreviations.

In the following summary the letters A, B and C represent names of variables, X, Y and Z represent any legal NODAL expression and Ln a NODAL line number.

<table>
<thead>
<tr>
<th>Command</th>
<th>Example of form</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASK</td>
<td>ASK A B C</td>
<td>NODAL types a colon for each value required; the user types a value to define each variable</td>
</tr>
<tr>
<td>CALL</td>
<td>CALL SUB(X,A)</td>
<td>Transfers control to subroutine with parameters X (value) or A (reference)</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>DIM A(3,2)</td>
<td>Dimensions a real array</td>
</tr>
<tr>
<td>DIMATION</td>
<td>DIM-I B(10)</td>
<td>Dimensions an integer array</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>DIM-S C</td>
<td>Dimensions a string array</td>
</tr>
<tr>
<td>DEFINE</td>
<td>DEF-FUN INJ:VB</td>
<td>Define NODAL text to be a subroutine, function, or string function</td>
</tr>
<tr>
<td>DEFINE</td>
<td>DEF-CALL NAME(R-X)</td>
<td></td>
</tr>
<tr>
<td>DEFINE</td>
<td>DEF-STR FCHR(S-X)</td>
<td></td>
</tr>
<tr>
<td>DO</td>
<td>DO 4.1</td>
<td>DO line 4.1; return to command following DO command</td>
</tr>
<tr>
<td>DO</td>
<td>DO 4</td>
<td>DO all group 4 lines or until RETURN is encountered. Return to command after DO Command</td>
</tr>
<tr>
<td>DO</td>
<td>DO X</td>
<td>DO as specified by value of expression X</td>
</tr>
<tr>
<td>EDIT</td>
<td>EDIT 4.1</td>
<td>Prepare for use of line edit commands.</td>
</tr>
<tr>
<td>EDIT</td>
<td>EDIT 4.1,L</td>
<td>If &quot;,L&quot; present, list specified line before Editing.</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
<td>Terminate program execution</td>
</tr>
</tbody>
</table>
APPENDIX 6
COMMAND SUMMARY

ERASE  ERASE 1.2 A B  Erase elements from local program storage.
ERASE  ERASE ALL  Erase all variables
EXECUTE  EX (X) 2 A  Sends group 2 of the user program together with element A of the program variable list for execution in computer X
FOR  FOR A = X,Y,Z;  --- Rest of line (including more FOR commands)
     FOR A = X,Z;  --- is executed with values of A ranging from X to Z by increments of Y. Y is assumed 1 if not present.
GOTO  GOTO 3.4  Transfers control to line 3.4 or line given by expression X rounded to 2 places after point
     GOTO X
IF  IF X>Y;  --- DO rest of line after ";"
     IF X<Y OR Y>Z;  --- if condition satisfied
     Conditions are:
            =  <   <> < =  > =  <>
     IF (X)Ln,Ln,Ln  Transfers control to first, second, or third line number according to whether X is negative, zero or positive.
     IF(X) Ln,Ln;  Transfers control according to conditions satisfied, otherwise continues with commands after.
     IF(X) Ln;

IMEX  IM(5) TYPE BCT(3)  Send command to computer 5 for immediate execution. Result typed out on terminal where IMEX command given

LDEF  LDEF FILE1 FILE2  Loads defined function area from files FILE1 and FILE2

LIST  LIST  LIST 1.1  NODAL types out entire program.
     LIST 1  Types out line 1.1
     LIST 1.1  Types out all group 1.
     LIST 1.1 2 3  Types line 1.1, group 2 and 3
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>LOAD FILE</td>
</tr>
<tr>
<td>OLD</td>
<td>OLD FILE</td>
</tr>
<tr>
<td>OPEN</td>
<td>OPEN FUNCTN</td>
</tr>
<tr>
<td>OVERLAY</td>
<td>OVE FILE X,Y</td>
</tr>
<tr>
<td>QUIT</td>
<td>QUIT</td>
</tr>
<tr>
<td>REMIT</td>
<td>REM A B</td>
</tr>
<tr>
<td>RETURN</td>
<td>RETURN</td>
</tr>
<tr>
<td>ROF</td>
<td>ROF</td>
</tr>
<tr>
<td>RUN</td>
<td>RUN</td>
</tr>
<tr>
<td></td>
<td>RUN FILE</td>
</tr>
<tr>
<td></td>
<td>RUN [X] FILE</td>
</tr>
<tr>
<td>SAVE</td>
<td>SAVE FILE</td>
</tr>
<tr>
<td></td>
<td>SAVE FILE 2 A B</td>
</tr>
<tr>
<td>SDEF</td>
<td>SDEF FILE</td>
</tr>
<tr>
<td>SET</td>
<td>SET A = X</td>
</tr>
<tr>
<td>TYPE</td>
<td>TYPE A+B-C</td>
</tr>
<tr>
<td>TYPE STR &quot;DEF&quot;!</td>
<td>Types out list of items which can contain expressions, strings, and control sequences, such as ! for carriage return line feed # for carriage return %X for format change to X &amp;X for X spaces.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>VALUE</td>
<td>VAL X</td>
</tr>
<tr>
<td>WAIT</td>
<td>WAIT (X)</td>
</tr>
<tr>
<td></td>
<td>WAIT-TIME X</td>
</tr>
<tr>
<td></td>
<td>WAIT-CYCLE 4</td>
</tr>
<tr>
<td>WHILE</td>
<td>WHILE A&lt;2; DO 3</td>
</tr>
<tr>
<td>ZDEF</td>
<td>ZDEF</td>
</tr>
<tr>
<td>$ASK</td>
<td>$ASK&quot;STRING IS&quot;A</td>
</tr>
<tr>
<td></td>
<td>$ASK A(5)</td>
</tr>
<tr>
<td>$DO</td>
<td>$DO STR1</td>
</tr>
<tr>
<td>$IF</td>
<td>$IF STR&gt;&quot;DEF&quot;; --- $IF (STR=&quot;DEF&quot;)Ln</td>
</tr>
<tr>
<td>$MATCH</td>
<td>$M STR1 P1 : X</td>
</tr>
<tr>
<td>$PATTERN</td>
<td>$PAT P1 = &quot;XYZ&quot;</td>
</tr>
<tr>
<td>$SET</td>
<td>$SET A=&quot;QWER&quot;</td>
</tr>
<tr>
<td></td>
<td>$SET A=INPUT(X):X</td>
</tr>
<tr>
<td>$VALUE</td>
<td>$VALUE &quot;XYZ&quot;</td>
</tr>
<tr>
<td>%</td>
<td>% COMMENT</td>
</tr>
<tr>
<td>?ON</td>
<td></td>
</tr>
</tbody>
</table>