LAB II-CO/74-2

December 1, 1974

EUROPEAN ORGANISATION FOR NUCLEAR RESEARCH

THE NODAL SYSTEM FOR THE SPS - 1974

M.C. Crowley-Milling, J.T. Hyman and G.C. Shering

ABSTRACT

This report groups together information on the Nodal software system. It contains a programming manual and gives information on how to write machine code modules for NODAL. In addition it gives an outline of the system design which should be useful for those who are studying how to use NODAL to control the various parts of the accelerator.

MAIN CONTENTS

APPENDICES

DETAILED CONTENTS

DETAILED CONTENTS CONTINUED
---

1. Introduction

This report describes the NODAL system for the SPS as it exists at the end of 1974. The report covers two main areas : firstly the NODAL language, secondly the NODAL system.

The NODAL language is a high level programming language with special features for use in real-time control, multi-computer applications, and interactive applications. It is based on FOCAL and SNOBOL4, with some influence from BASIC.

The NODAL system is the way in which NODAL is implemented and used on the basic hardware and software of the SPS computer control system. The hardware consists of 24 NORD-10 computers linked together via a serial link message transfer system involving another NORD-10. The basic software consists of an extended version of the NORD-10 SINTRAN II operating system, called SYNTRON, which incorporates a multi-computer filing scheme.

Not all the facilities described in this report will be included in all computers. In particular NODAL defined functions (chapter 8) and string handling facilities (chapter 9) will only be available in the console computer, and the advanced pattern matching string processing facilities (chapter 9) will only be available in the service computer.

Chapter 2 describes basic NODAL facilities which can be used at any interactive terminal at any computer. The beginner is advised to read this chapter carefully then go to a terminal on the TSS system and try to write a few simple programs.

2. Basic NODAL Programming

2.1 Simple commands TYPE and SET

NODAL is an interactive interpretive system and so is used by typing directly on-line at the control device, either a Teletype or a VDU (cathode ray tube display plus keyboard). The smallest useful element of NODAL is the command. 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 1.

2.2 Numbers, variables and expressions

The command body syntax is made up mainly of numbers and variables, often combined into expressions.

2.2.1 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

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.

2.2.2 Octal Integers

Integers in the range -32768 to 32767 can be represented by a string of octal digits (0-7) preceded by "[", for example

[177777

represents the integer -1.

2.2.3 RADIX36 Integers

Integers in the range -32768 to 32767 can also be represented as 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.

2.2.4 Variables

A variable 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. Example 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.

2.2.5 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 substraction (-), 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 operatons, 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
```

2.3 Lines and the FOR command

The most important subdivision of a NODAL program is the line of text. No action is taken by NODAL until the carriage return key <CR> is pressed on the keyboard. The line is only then read in and action taken. A line can contain several commands separated by semi-colons, e.g.

```
SET A=1; SET B=2; SET INJPHS = 2*A+5*B
```

2.3.1 The FOR command

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

```
FOR I=10,20; SET VACVLV(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 say A, B, C and has the form

```
FOR I = A,B,C;...
```

where I initially takes the value of A, incrementing by B through C. If B is not present it is assumed to be unity. A, B and C can be fractional or negative if desired.

2.4 The CALL command

This command is used for calling subroutines and has the syntax

```
CALL NAME(PARA 1,...PARA N)
```

where the parameters can be of types numeric value, string value, or reference. E.g. the command

```
CALL TT10ST(10)
```

might set the beam line TT10 to 10 GeV. Normally subroutines will be used to take values from, or return values to the main program. E.g.

```
CALL COACQ(PLANE, TIMING, TABLE)
```

The first two parameters could be value parameters defining the plane and timing of a closed orbit acquisition, TABLE could be a reference parameter defining the array into which the results are to be put.

Subroutines can be written in NODAL (chapter 8) and in assembly language (chapters 6 and 7). In the console computers most process subroutines will be written in NODAL whereas in the local computers most will be written in assembly language.

Use of the keyword CALL is optional. Thus one can say

```
NAME(PARA1, ... PARAN)
TT10ST(10)
COACQ(PLANE, TIMING, TABLE)
```

2.5 Program sequencing

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.

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 given time, or at regular intervals of time.

2.5.1 RUN command

This command starts program execution. RUN on its own starts execution of a program just typed in, starting at the lowest numbered line.

RUN [2.7]

however, starts execution at line 2.7 . (Note 2.7 = 2.70)

RUN FILENAME

causes the program stored in FILENAME to be loaded and executed.

RUN [X] FILENAME

causes the program stored in FILENAME to be loaded and execution to start at the line number given by the value of expression X.

2.5.2 DO command

The DO command is used to transfer control to a specified line, or group, and then return automatically to the command following the DO command. For example the program :

```
1.1 SET A=1; SET B=2
1.2 TYPE "STARTING = "
1.3 DO 3.2

2.1 TYPE "FINISHED = "

3.1 SET A=3; SET B=4
3.2 TYPE A+B
```

gives on typing RUN

```
STARTING =     3       FINISHED =     7
```

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 acquisiton 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. This feature gives a powerfull computed "CASE" type facility.

2.5.3 RETURN command

The RETURN command is used to exit from a DO subroutine. 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.

2.5.4 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 .

It is not possible to jump out of a DO group using the GOTO command. Any such attempt will be treated as a RETURN command. Thus 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.

2.5.5 The END and QUIT commands

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 interrupt programs END causes a return to monitor and the program disappears. QUIT always causes a return to monitor and NODAL is disconnected.

2.5.6 % command

Beginning a command string with the symbol % 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.

2.6 Making decisions, IF and WHILE

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 or WHILE commands.

2.6.1 The arithmetic IF

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.

IF commands can be coupled together to get complex conditions, e.g.

```
2.1 IF (A)3.1; IF (10-A)3.1,3.1; DO 4
```

In this example group 4 is executed if $0 \leq A < 10$, otherwise control is transferred to line 3.1.

The expression in brackets is taken to be zero if its magnitude is in the range $\pm$ 10E-6.

2.6.2 The logical IF

This allows the execution of the remainder of a line of commands to be conditioned 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 arithmetic IF above. The two expressions are considered equal if the difference between them is less than $\pm$5E-8 relative to the first.

2.6.3 The WHILE command

This is similar to the logical IF 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.

2.7 System variable subroutines

NODAL is linked to the hardware of the accelerator via machine-code subroutines. These subroutines can take two forms : system variables, and CALL subroutines.

System variables are at the heart of the NODAL on-line features. They allow machine parameters to be handled with the same power and flexibility as ordinary program variables. Control or acquisition goes through a user-written subroutine, however, instead of simply to computer memory. To avoid filling the computer memory with many similar routines one will make use of parameters. For example one system variable 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 consideraton. So

```
TYPE VPS(40,#CUR)
```

might access the current in pump 40

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

might switch pump 40 on. System variables 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

```
TYPE VPS(40,#CUR)
```

could access the true pump current, whereas

```
TYPE VPS(40,#CUS)
```

might access the last scan value.

CALL subroutines provide a more general link between the NODAL program and the hardware. They are called as

```
CALL NAME (PARAMETER 1, ...)
```

The parameters or arguments of the above CALL or system variable routines can be of three types : numerical value, string value, or reference. Numerical value parameters are used to pass a number to a subroutine. In

the calling sequence the parameters are used to pass a character string to the subroutine, and take the form of a string expression. Reference parameters are used to pass the address of NODAL data elements (e.g. arrays) to the subroutine. In the calling sequence a NODAL name is used. For example, with

```
CALL NAME(3, "YES", ABC)
```

the subroutine NAME gets the number 3, the string YES, and the address of the array ABC.

2.8 More about Input and Output

2.8.1 TYPE command

This command was introduced in section 2.1 and has been further illustrated in several of the examples.

The general form of the command is

```
TYPE typelist
```

where "typelist" is a series of primary elements. Primary elements are expressions for which the value is printed, strings e.g. "THIS IS" for which the text between the quotes is printed out, and control sequences such as ! for a new line. For instance :

```
TYPE "VALUES ARE" A B ! "END"
```

could result in the type-out

```
VALUES ARE   2.5362   4.1234
END
```

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
\X	Type ASCII equivalent of X
&X	Type X spaces
]X	Type out X in octal
?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. Including the control sequence %8.03 in the TYPE list 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 1.00000000E0. Note that this use of % occurs only in a TYPE list, 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 feld of 5.

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

```
TYPE A B
```

is the same as

```
TYPE A,B
```

but

```
TYPE A -B
```

is not the same as

```
TYPE A, -B
```

2.8.2 ASK command

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

```
11.99 ASK X Y Z
```

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 is asked for, e.g. X Y Z in the example above, all three values can be typed on the one line separated by commas.

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 execute line 11.99 first of all is typed

```
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.

2.9 Errors, LIST and ERASE, trace facility

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.

2.9.1 Error detection

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

```
NONEXISTENT NAME 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 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.

2.9.2 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.

2.9.3 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.

2.9.4 The Trace feature

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.

2.10 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, i.e. performs an automatic ERASE.

The EDIT command, however, provides powerful facilities for editing lines, both old lines already in the computer and also new lines as they are being typed in. This facility is identical to the line edit facility in QED, the NORD-10 editing program.

The EDIT command has the following format

EDIT NN.NN ,L

NN.NN is the line number of the line to be edited. The L denotes a required listing of the line before edit mode is entered and is optional. The EDIT command works by mapping the old line, NN.NN, onto the new line which is being created. Special characters are used to control this 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. EDIT is normally in the replace mode so any character other than a CTRL character is entered in the old line. For example to edit the line.

1.1 LET A=1

which should be

1.1 SET A=1

one can type EDIT 1.1 then 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 id echoed as >. For example suppose we type EDIT 1.1, L which gives

1.1 SET A=1

and we want

1.1 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 much used, 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 II.

2.11 Arrays and Mathematical Functions

2.11.1 Arrays

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

```
DIM  B(10,2)

DI-I J(4,5)
```

Strings arrays are created using the DIMENSION-STRING command e.g.

```
DIM-S  STR
```

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 boundares to prevent writing outside the array by programming error.

2.11.2 Mathematical functions

These have the usual meaning and are really just system variables of a mathematical rather than accelerator type. For instance

```
SET Y = SIN(X)
```

has an obvious meaning. The functions available are

Function name	Value returned
SIN(X)	Sine of X, X in radians
COS(X)	Cosine of X, X in radians
AT2(X,Y)	Arctangent of Y/X in range 0 - 2 PIE
SQR(X)	Square root of X
EXP(X)	Exponential of X
LOG(X)	Natural logarithm of X (LOGe(X))
ABS(X)	Absolute value of X
INT(X)	The integer part of X
FPT(X)	The fractional part of X (same sign as X so that INT(X) + FPT(X) = X)
SGN(X)	±1 according to sign of X
MOD(X,Y)	X modulo Y

3. 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. This could range from a number e.g. 100 in a basic SINTRAN system, through a NODAL name as in the temporary system described in LAB-CO/INT/Comp.Note/74-2 to a full TSS file name. On other types of computer it would be the appropriate file specifier, e.g. DT1:FRED on the PDP-11 DOS.

3.1 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, e.g.

SAVE FILENAME ABC

will save the array ABC on the file FILENAME. Programs and data can be saved together, e.g.

SAVE FILENAME 2 ABC

will save group 2 of the current program, plus the array ABC on the file FILENAME.

Material which has been saved can be retrieved using the LOAD command, e.g.

LOAD FILENAME

The contents of FILENAME are loaded and incorporated into the current program. If FILENAME contains lines of program or data elements of the same name as those already existing in the program, the already existing entries are deleted and the new ones inserted.

The OLD command is an effective ERASE ALL followed by a LOAD, e.g.

OLD FILENAME

clears the current contents of the user area then loads the contents of FILENAME.

The RUN command described in section 2.4 above is an effective OLD followed by RUN.

3.2 Task global variables

In small computer configurations the NODAL buffer may be too small to hold the complete program. One way to get around this problem is program chaining. The program is divided into several sequential parts each stored on a separate file, e.g. FILE1, FILE2, 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 however. 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.

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, ie. are not affected by ERASE ALL, OLD, SAVE, LOAD etc. They are set or read as

```
SET ARG(5) = X

SET X = ARG(5)
```

This allows sequences of chained "RUN FILE" programs to be easily set up, the main interchange parameters being held in ARG(1)... ARG(16).

3.3 Sub-programs on files

Several levels of sub-program structure are provided in NODAL. Separate named subroutines are, of course, provided and these are discussed elsewhere. However lines and/or groups in NODAL can be used as subroutines, e.g.

```
1.1 DO 2.1; DO 3; DO X; QUIT
```

is a line which makes three subroutine calls, to line 2.1, group 3, then the line or group corresponding to the value of X.

It is often convenient to be able to write and store sub-programs separately. This can be done using the SAVE and LOAD commands discussed in the previous section, as a line, or group, can be saved on a file, then loaded and executed with a DO command. This is more a dynamic re-configuration of the main program, however, rather than the call of an independent sub-program.

A typical sequence for this use might be

```
1.1  % MAIN PROGRAM TO LOAD AND EXECUTE GROUPS FROM FILES
1.2  LOAD FILE1 ; DO 10 ; SET A=3 ; DO 11
1.3  ERASE 10 11 12 ; LOAD FILE2 ; DO 30 ; END
```

This sequence loads groups from files then executes them with DO statements. The LOAD command is normally meant for loading data elements but can also be used, as above, for loading program material into the buffer and mixing it with what is already there. The lines loaded during execution of a program must, however, be higher than any currently active line.

3.4 The OVERLAY facility

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 buffer with the main program so their sum of their lengths must fit into the buffer. 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 task buffer. 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) as described in section 3.2. 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.

4. REAL-TIME FACILITIES

As NODAL is a language for real-time control of the accelerator, facilities are provided to synchronise programs with the accelerator timing or with clock time. Programs can also be scheduled to run at given times or hooked to interrupts.

4.1 The WAIT Command

Programs are synchronised by means of the WAIT command. This has three forms :

WAIT-TIME TIME

WAIT-CYCLE EVENT,DELAY

WAIT (computer number)

The first causes the program to wait for a given number of seconds before going on to the next instuction. 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 ommitted if not required.

The third form causes the computer to wait for data which is expected to be remitted from another computer. This is discussed further in section 5.

I M P O R T A N T N O T E

Use of the first two forms of the WAIT command can cause the program to lock up valuable resources in the computer for a long time. They should not be used in high priority "real-time" programs, or in a sequence of commands sent to a remote computer for execution by means of the EXECUTE command (see section 5).

4.2 Program Scheduling

NODAL programs can run on three "NODAL levels". These are the "interactive" level, the "real-time" level, and the "immediate-command" level. These should not be confused with operating-system levels, the relationship between the two depending on the particular system.

The interactive level is that which has been discussed up to now. Programs at this level run in conjunction with an on-line teletype in direct liaison with an operator. Programs from files can be started using the RUN command. They can get reponse from the operator using the ASK command.

The real-time level is for a different type of program. It is for programs which have been written, tested and stored at the interactive level but which are required to run autonomously. Their running is controlled by time or interrupt rather than an operator and their only connection with an operator is to type out a result, or more likely a warning message. Real-time programs could be scheduled to check some currents or pressures every 15 minutes say, or to acquire some data every cycle and store it on a file.

The immediate command level is discussed later.

Programs stored on files can be scheduled for real-time operation by the following commands :

```
SCHEDL(FILE,ODEV,TIME,INTERVAL)

RTRUN(FILE,ODEV,EVENT,DELAY)

REPEAT(FILE,ODEV,EVENT,DELAY)

HOOK(FILE,ODEV,LAM NO.)
```

SCHEDL causes the program stored on FILE to be started at time TIME and repeated every INTERVAL (not repeated if INTERVAL $\leq$ 0). ODEV specifies the device on which output can occur. TIME and INTERVAL are specified in seconds.

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.

The command

DISCON(FILE)

disconnects the program on FILE 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" in 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 calender specifications to seconds and vice-versa. The function

LISP

causes NODAL to print out the state of the programs in the computer, ie. it tells which programs are HOOKed, SCHEDLed, etc.

5. Interaction Between Computers

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.

5.1 The IMEX command

IMEX, for immediate execute, causes a command to be obeyed in a remote computer and any result to the 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. In fact the remainder of the line is sent down so more than one command can be transmitted. A typical sequence might be

```
>IM (5) TYPE BCT(3,#TIM) BCT(3)
  2563   59.6
>
```

5.2 The EXECUTE command

This is the main command for programmed interaction. It is very similar to the SAVE command described in chapter 3, except that instead of saving NODAL elements on a file it sends them to another computer for execution. For example

```
EXECUTE (5) 2 ABC
```

will send group 2 and the array ABC to computer 5 for execution as a real-time task. 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 are legal for execution.

5.3 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

```
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 program or sub-program which did the EXECUTE. The REMIT command is the last command executed in the EXECUTE task, ie. it contains an implied END.

5.4 The WAIT command

The WAIT command in its form

```
WAIT (5)
```

causes data REMITted from an EXECUTE task to be read into the task buffer. 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, otherwise when the program returns to command mode.

5.5 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
```

6. Writing Machine Code Modules for NODAL

NODAL has been described in the past as a "subroutine caller". Certainly the power and speed of any NODAL system depends largely on the repertoire of machine code subroutines available. Such subroutines can be called by NODAL in one of three ways, viz.. with the call statement,

```
CALL FFT(A,-1)
```

or read functions, e.g.

```
SET Z=CAMAC(C,N,A,F)
```

or as write functions, e.g.

```
SET MBBH(1,5) = ⊂17
```

Provision of the last two modes makes NODAL a true control language as features of the accelerator can be accessed, via the machine code routines, in exactly the same way as program variables or arrays. Eleven of the NODAL element types (see appendix 6) are concerned with machine code modules. These are

Type 6 mathematical function

```
e.g.  SET Y = SIN(X)
      SET Z = MOD(X,Y)
```

These are read functions with either one or two real value arguments.

Type 7 System variables

```
e.g.  SET MBBH(1,5) = 217
      SET Z = MBBH(1,7)
```

These are read-write functions with zero, one or two integer value arguments.

Type 8 read-write functions

```
e.g.  SET CAMAC(C,N,A,F) = 512
      SET Z = CAMAC(C,N,A,F)
```

These are read-write functions with up to 8 arguments of a wide range of different types which will be described

Type 9 write-only functions

The write function subset of type 8

e.g. SET VLT = 50

Type 10 read-only functions

The read-only subset of type 8

e.g. SET Z = BUTTON(2)

Type 11 Call subroutine with NODAL error return

e.g. CALL SUBR(A,1+B,3,"TEXT")

Up to 8 general purpose parameters as in type 8.

If an error is detected the subroutine can cause NODAL to stop and print out an error message (this facility is also available in the preceding types).

Type 12 Call subroutine with no NODAL error return

As type 11 but wihout the possibility of triggering a NODAL error message. This type is included to cope with assembly language or Fortran subroutines which already exist.

Type 18 Free routines

These can be read/write functions or CALL subroutines. They are essentially NODAL routines and must use the internal facilities of NODAL to get their working space, parameters, etc.

Types 22, 23, 24 String functions

These are discussed further in the chapter on string facilities.

GENERAL PURPOSE MODULES

Type 6 is specialised for calling FORTRAN LIBRARY routines and is not described in detail here. Type 6 is very important and is described fully in chapter 7. Types 18, 22, 23, 24 are described later. The rest of this chapter discusses the general purpose FORTRAN compatible types 8, 9, 10, 11, 12.

6.1 Parameter passing - SINTRAN standard call

NODAL makes the call to the machine code subroutine in the SINTRAN standard way. That is with the A-register pointing to a list of the parameter addresses. This provides compatibility with existing SINTRAN routines and with FORTRAN. The parameters supplied by NODAL to the routine consist of the "explicit" parameters in the NODAL program plus some "implicit parameters". In types 8, 9 and 10 two implicit parameters are supplied, preceding the explicit parameters, e.g. the JPL made in

```
SET CAMAC(1,2,3,4) = 5
```

has 6 parameters. The first is the VALUE, a three word floating point parameter which will have the value 5 in the above example. The second is the flag, an integer which has the value -1 in the above example, signifying that this read-write function is being used in the write mode. It would be +1 in the read mode, e.g. in

```
SET Z = CAMAC(1,2,3,4)
```

In this case VALUE will have an indeterminate value at entry to the subroutine but must contain the result of the read at exit. At exit, the routine should set FLAG to zero if all is OK, otherwise to an error number when NODAL will stop and print out an error message. NODAL has some "canned" error messages as listed in appendix 5. Use of the corresponding error number causes the error message to be printed out, for instance

```
ARGUMENT LIST ERROR AT LINE 1.10
```

If the error number is higher than those corresponding to "canned" messages, then NODAL types out the error number, e.g.

The "implicit parameters" are then followed by the explicit parameters, in the above example 4 integer values.

In type 11 CALL subroutines only one implicit parameter is supplied, the FLAG. This is set to zero at entry and at exit has the effect described above.

In type 12 no implicit parameters are supplied, so the parameter list pointed to by the A-register corresponds exactly to the "explicit parameters".

6.2 Working space

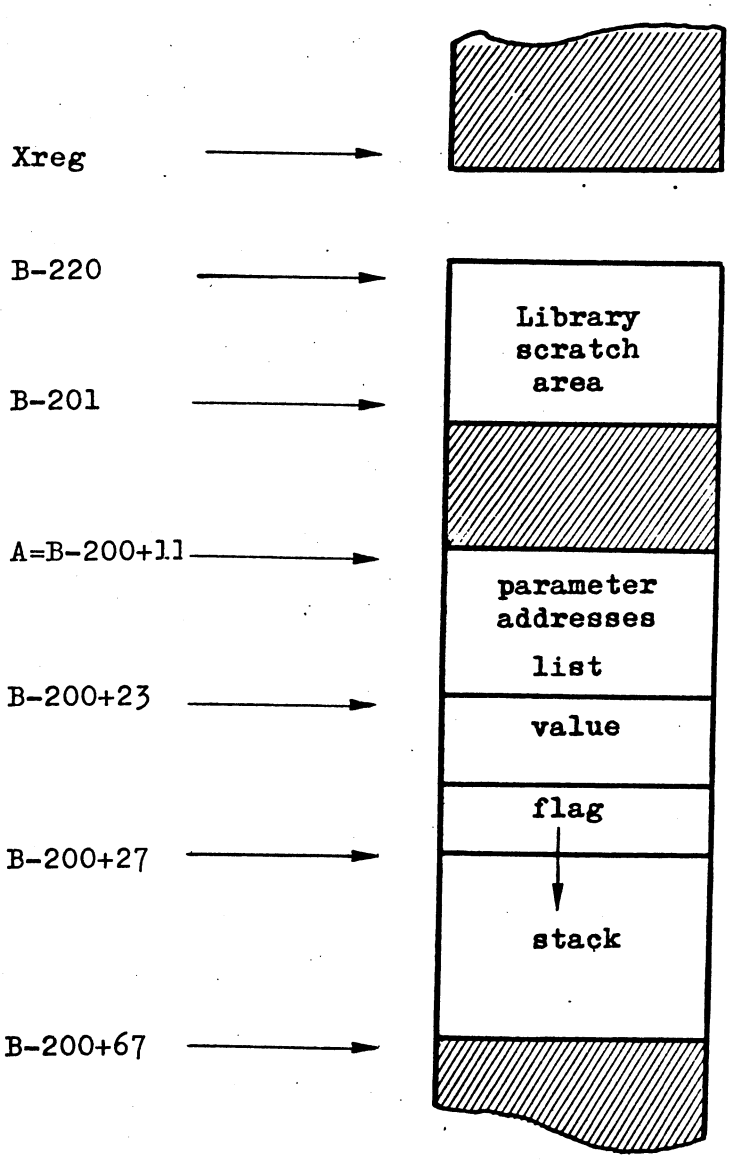
One of the features of the NORD-10 NODAL-SINTRAN system is the use of re-entrant code. NODAL subroutines should be re-entrant so that they can be used on several levels. Any working space required by these routines should be obtained from the system.

At the lowest level this can be obtained from SINTRAN using the routines PUSH and POP. This is normally done if the routines are called from assembly language programs or other routines. Use of PUSH, however, runs the risk of putting the program into the "waiting state" if it executing whilst another higher priority program is in the "waiting state".

Routines called only from FORTRAN or NODAL can use the library scratch area located from B-220 to B-201 (octal).

Routines callable only from NODAL can use other facilities which are out-lined in the figure below.

Stack_layout_at_call_from_NODAL



all numbers in OCTAL

In addition to the library scratch area in B-220 to B-201 NODAL offers further working space. The X-register points to the last word in the task buffer currently being used so all the space between this and B-201 is free. Also NODAL uses a stack area of 40 octal words to hold value parameters (up to 8 string descriptors of four words each). This is rarely all used so this area can be used by the user.

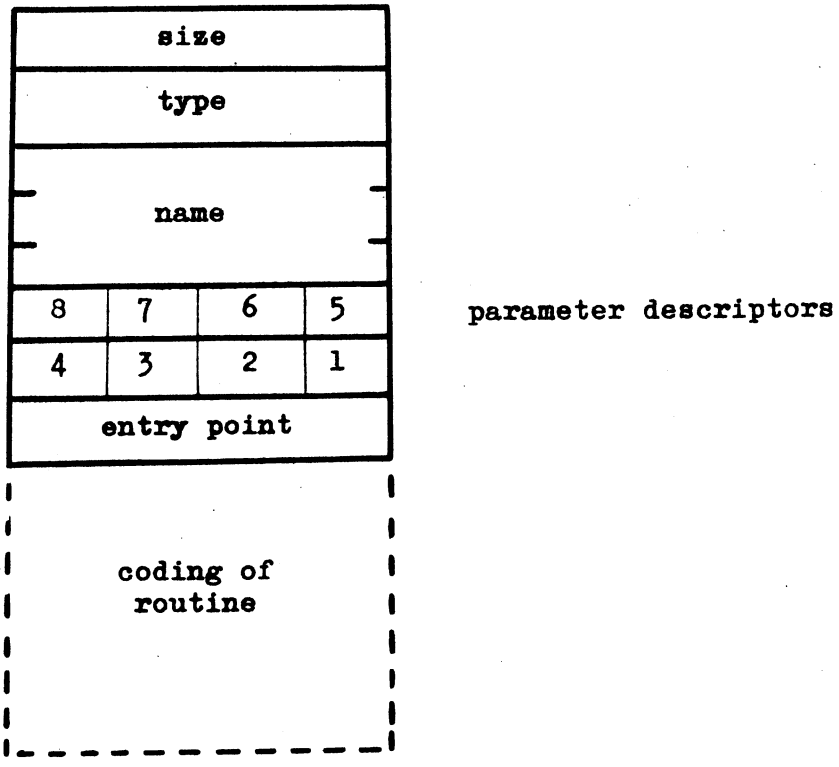
In NODAL routines the parameter addresses can be accessed directly via the B-register. Also value parameters can

themselves be accessed directly via the B-register as each value parameter is stacked one after the other in the stack area shown in the diagram. An example of a routine using these facilities is given in Appendix 3.

Linking to NODAL - the header

In order to call the routine, NODAL must be given some information such as the name, the type, the parameter specifications and the start address. These are contained in the "HEADER" which has the form shown below.

Layout of HEADER



NODAL HEADERS must be included in some NODAL named list which can also contain other NODAL named elements (e.g. DATA arrays, NODAL subroutines etc.). Each entry in a named list must start with the size of that entry as this is the pointer to the next entry since they are put end to end.

The size must be positive and the list terminated by -1. It is frequently convenient to include the body of the routine in the entry with the header as shown by the dotted line in the diagram. For the machine code routines discussed here the type will be 8, 9, 10, 11 or 12.

The NAME consists of 6 consecutive bytes, unused being zero. The entry point is the start address of the routine.

6.3 Parameter descriptors and types

Up to eight parameters can be specified by the two parameters 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 held in stack
2	integer value	- 1 word integer held on stack
3	string value	- 4 word string descriptor held on stack
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 on stack

NOTE The address of the parameter is contained in the list starting at B-200+11 pointed to by the A-register. The

actual parameter itself is elsewhere, though for value parameters these are contained in the parameter stack as discussed earlier.

6.4 Calling sequences

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

```
SET CAMAC(SIN(PIE/Z),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 mal-function at present if the same name is used in response to more than one type 6 specification in the same routine.

NOTE Normal EXIT via the L-register should be made at the end of the routine. The B-register must have the same value as at the entry.

7. Data Modules

The concept of system variables and data modules has been described in Lab II-CO/Int/00/73-8. This chapter describes in detail the rules for calling data modules and linking them to NODAL through a type 7 named entry in a NODAL element list.

7.1 Assembly language calling sequence

Data modules are called in the Sintran standard manner. At entry the A-register points to a list of parameter addresses and the routines can use a scratch area in B-220 to B-201.

Four parameters are used. These are:

a. VALUE (three word floating point). This contains the data to be written to the equipment, or returned after reading.

b. FLAG (integer). This indicates whether the operation is a read or a write. NODAL supplies +1 for a read operation, -1 for a write. It has been suggested that other values, eg. +2 and -2 could be used by tested machine code progams to bypass some of the routine checking. On return FLAG must be set to zero if all is OK, otherwise to an error number.

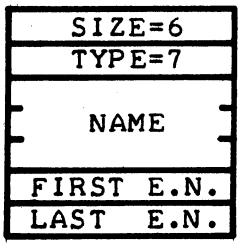
c. EQUIPMENT NUMBER (integer). This indicates the particular equipment in question. The format of the equipment number is described in the first reference given above.

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

The data module will normally check the parameters for legality. In many cases it will also want to check the access priority of the calling program, ie. the password of the user. This is not passed as an explicit parameter, but can be obtained by a call to the routine "PASWD" which will return the user password.

7.2 Communication with NODAL

This is ensured by means of a type 7 named entry in a NODAL user list, together with a 64 word table containing the entry points of the data modules. The header is shown below



Thus 7 words are needed for every name which points to a data module. The equipment number contains 6 bits indicating the equipment type. These are used by NODAL to branch to the correct subroutine via the table SVTAB in core. This might look as follows

```
SVTAB, EQT0
       EQT1
       --
       --
       EQT63
```

where EQT0, -- EQT63 might be the entry points of the 64 data modules. For computers with a drum, a second table CLTAB is used. This contains the number of the core-load on which the module resides.

The serial number part of the equipment number indentifies the particular unit referred to by that name. If several equipments are sharing the same name, eg. VPS(1), -- VPS(50), then the serial numbers are those of the first and last units of that name.

7.3 NODAL calling sequence

The basic calling sequences, say for a system variable MBBQ, are

```
SET MBBQ(1,#CUR)=456
SET A=MBBQ(1,#CUR)
```

In the first example the routine is called by the NODAL setting subroutine, and in the second by the expression evaluating routine. Thus the system variable read value can be used in any NODAL expression, eg in TYPE, IF, FOR, commands.

The property specifications can be ommitted, for example the command

```
SET MBBQ(1)=456
```

will be accepted by NODAL, and a property number of zero will be given to the data module. The data modules should interpret a zero property specification as being the most commonly required property, eg. current for a power supply.

The unit number can also be ommitted where it is not required, eg.

```
SET MBBH=456
```

is a meaningful command if there is only one MBBH. The equipment number transmitted is identically that contained in the header for the name MBBH.

7.4 An example

Appendix 4 is a cross-reference listing of the assembly language source code for an example system subroutine set-up. Lines 10, 11 and 12 specify three headers MBBH, MBBV and MBBQ. These headers are contained in some NODAL list of named elements, the -1 terminator being shown in line 12. (NODAL lists are discussed further in chapter 11). The equipment number in these type 7 elements is made up of equipment of type 2, serial numbers 100 (octal 144), 101, 102.

The start address for data module 2 is found by NODAL as the third entry in the address table SVTAB, here EQT2. The rest of the example shows the data module itself.

This subroutine requires two scratch variables, SERNO to store the serial number, and RETAD to store the return address. These are defined in lines 26 and 27, with reference to the new value of the B-register as shifted by -20 in line 32. First of all the equipment number is checked in line 31 to see that it is indeed of type 2. This is not necessary when called from NODAL, but could perhaps be wise if the routine is to be called from FORTRAN or assembler programs. Then in lines 32 and 33

the serial number is checked to see if it is in range. If not the error number 35 (ILLEGAL EQUIPMENT NUMBER) is given. Then the property is checked for legality, if not correct then error 36 (ILLEGAL PROPERTY) is given. If property=0 then the default property, here 42, is used. No checking for password is done here. Finally FLAG is checked and the appropriate read or write operation performed. Here the reading or writing is to the simple data-table DTB2 defined in line 50.

8. NODAL DEFINED functions and subroutines

The operation of NODAL is based on its ability to call named functions and subroutines. Up to present only subroutines written in assembly language have been discussed. This chapter describes how named procedures can be written in NODAL.

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, ie. in an immediate command. Thus defining a procedure is a separate exercise from writing a main program. Chapter 6 and 7 describe how to write procedures in assembly language, this chapter does the same for NODAL defined procedures. All procedures, once defined, become part of the system and so are external to the main program.

In the local computer it is envisaged that most procedures, e.g. Data-modules, will be written in assembly language to provide the link between the computer and the hardware. In the console computer, however, links to the hardware are achieved by sending NODAL commands or programs over the data-links and REMITting the results 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 creats 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.

8.1 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

```
DEFINE-FUNCTION NAME

DEFINE-CALL NAME

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". 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 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. These are set according to the algorithm used for data modules as described in chapter 7. 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.

If really no parameters are wanted then this can be forced as

DEF-FUN NAME()

8.2 The OPEN command

This command takes the text from the procedure and puts it into the text buffer. The procedure is deleted. It is thus the reverse of the DEFINE command and is used for editing. E.g.

OPEN NAME

8.3 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 incorporate an END command so cause exit from the function. For write functions or CALL subroutines the END command is used. Examples of these commands are

```
1.9  VALUE X+Z*Y

1.9  $VALUE "THIS IS THE RESULT STRING"
```

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

SET ERROR = X

were X is the error number. This causes NODAL to enter its error reporting mode as described in chapter 2, just as if the routine were a machine code routine as described in chapters 6 and 7.

8.4 Utility functions

The following user CALL type subroutines are available for managing defined functions :

LISD

SDEF FILENAME

LDEF FILENAME

ZDEF

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

SDEF causes the current defined functions to be saved on the file FILENAME.

LDEF causes the functions stored on FILENAME to be loaded for use.

ZDEF clears the current DEFINEd function list.

8.5 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.

9. String Facilities

NODAL is not only an interactive language but is also, at least in the console computers, a language for programming interaction. In this application the ability to handle character strings is very important. The string facilities offered by NODAL fall into two groups. Firstly string handling facilities. These allow creation, concatenation, sub-division, comparison, execution, input and output of strings. These facilities are available in the console computers. Secondly string processing facilities. These allow processing of strings by searching for patterns, replacing sub-strings by others and creation of new strings out of old strings. These facilities are restricted to the service computer. String variables and arrays are, of course, standard NODAL elements so could be sent from a console computer to the service computer for processing.

9.1 $SET command, string variables and arrays

String variables can be created by means of the $SET command, for instance 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"
```

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. Elements can then 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

```
TYPE FIND(A,"JOE")
```

would cause the number 10 to be printed out.

9.2 String functions

In previous chapters extensive reference has been made to arithmetic functions which return a numeric value when called, eg. SIN(X). The equivalent exists for strings, that is a function which returns a string when called. For example

```
$SET B = SUBS(1,5,A)
```

sets the string variable B equal to the string returned by the string function SUBS. Here SUBS returns the substring 1 to 5 of the string variable A, ie. the first five characters of A.

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

```
3.1 $SET A = INPUT(65):3.8; DO 4
```

the $SET command sets A to the string returned by the function 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, 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.

9.3 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 series of 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 chapter 2 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
```

i.e. 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. In the following notes the string "CONC" will be used to denote any legal concatenation as here described.

9.4 $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 variable can be asked for in one $ASK command, just as in the numeric ASK command.

9.5 $IF command

This command can be used to compare two strings. 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 comparision 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.

9.6 $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.

9.7 Some useful functions

Some useful string functions, or arithmetic value functions useful in string work are descrribed below.

STRING = SUBS(I,J,CONC)	returns substring I to J of concatenation
VALUE = SIZE(CONC)	returns number of characters in concatenation
VALUE = EVAL(CONC)	treats CONC as an arithmetic expression and returns its value
VALUE = FIND(ARRAY,CONC)	returns index of CONC in array, -1 if not found
VALUE = OPEN(CONC1,CONC2)	opens for read if first character in CONC1 is "R", or write if "W". Opens file of name given by CONC2, returns logical number.
CALL CLOSE(VALUE)	close file of logical number VALUE. All files if VALUE is -1.
STRING = INPUT(VALUE)	returns string from file of logical number VALUE, up to first carriage return line feed. Discards carriage return line feed, ie. picks first line off file.
STRING = INPUT	gets line from teminal (implied VALUE = 1)
STRING = INPC(VALUE)	gets one character from file
STRING = ALPHA	pre-defined string variable, returns A to Z
STRING = NUM	pre-defined string variable, returns 0 to 9

String functions can also go on the left of the equals sign in the $SET command, eg.

$SET STRFUN(...) = CONC

This is used for console devices such as self-scan units etc., in TSS as

OUTPUT(VALUE) = CONC	OUTPUT CONC onto file VALUE, then add carriage return line feed
OUTPUT = CONC	output line onto terminal
OUTC(VALUE) = CONC	output CONC without carriage return line feed

9.8 $MATCH and $PATTERN commands

These are the main string processing commands which can be used on the service computer. An example is

2.7 $MATCH STRING PATTERN : 2.1; DO 5

This matches STRING against PATTERN. 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 <CONC> PATTERN : 2.1

where STRING has been replaced by any valid concatenation enclosed in angle brackets.

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 = PATTERN

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

This section has described the syntax of the $MATCH and $PATTERN commands, but has skipped over what is meant by matching and what is a pattern. This is the main concept of string processing and is described in the next section.

9.9 Patterns and pattern matching

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

```
2.1  $MATCH A "YES" : 2.2; TYPE "INCLUDED"; RETURN
2.2  TYPE "NOT INCLUDED"
```

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 also contain replacement and assignment fields. For instance

```
$MATCH A "YES" = "*"
```

will search A for the string "YES" and if it finds it, it will replace it with the string "*". Thus if A were "YESSIR" it would become "*SIR". The assignment field causes the part of the string which is matched to be copied into a variable. For instance

```
$PAT P1 = "A" ("B" ! "C") "D"
FOR I = 1,10; $M A(I) P1 .OUTPUT
```

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

```
$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

```
$PAT P4 = 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

```
$SET A = "YES, YES, YESNO"
```

then

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

will produce the type-out

```
YES
YES
YES
YESNO
```

9.10 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.

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, ie. 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.

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) ,LABLE
```

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

BBEAK(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, ie. no other alternatives are tried.

10. Miscellaneous Functions

Many useful NODAL facilities are implemented as functions rather than as commands. Call functions, however, can look very like commands as the "CALL" directive is optional, eg.

```
LIST
```

lists the program and is a command, whereas

```
LISV
```

lists the variables and is in fact implemented as a function. Two differences must be noted however : firstly commands can be abbreviated whereas function calls cannot; secondly functions are optional in any given computer configuration so may be left out to gain space.

10.1 ODEV

This function causes the output stream to be switched from the normal terminal to the stream specified, eg. on TSS the sequence

```
SET ODEV = 5; LIST
```

will cause the program to be listed on output stream 5, ie. the line printer. At the end of an immediate command line, or program, ODEV is set back to the interactive terminal automatically.

10.2 ERROR

When a NODAL program encounters an error it stops and prints out an error message, as described in chapter 2. This operation can be provoked artificially using the command

```
SET ERROR = N
```

where N can be any positive integer >0. Some values of N correspond to particular "canned" error messages which are printed out as described in Appendix 5. Otherwise, ie. if N is greater than the maximum "canned" error message number, the message, eg.

```
ERROR 143 AT LINE 1.1
```

is printed out for N = 143 say. A special use of ERROR is described in chapter 8 for NODAL defined string functions.

10.3 LISV

This function lists out the user's variables and arrays, giving the values of variables and the dimensions of arrays.

10.4 LUST

This function lists out the machine code modules available to the user, together with their types and parameters.

10.5 HELP

This function lists the NODAL commands available.

10.6 BIT

This function allows manipulation of single bits in a 16 bit integer represented in the variable, say A in the calls

```
SET BIT(1,A) = 1
SET Z = BIT(15,A)
```

where the bit numbers are 0 to 15 and A must be a simple variable.

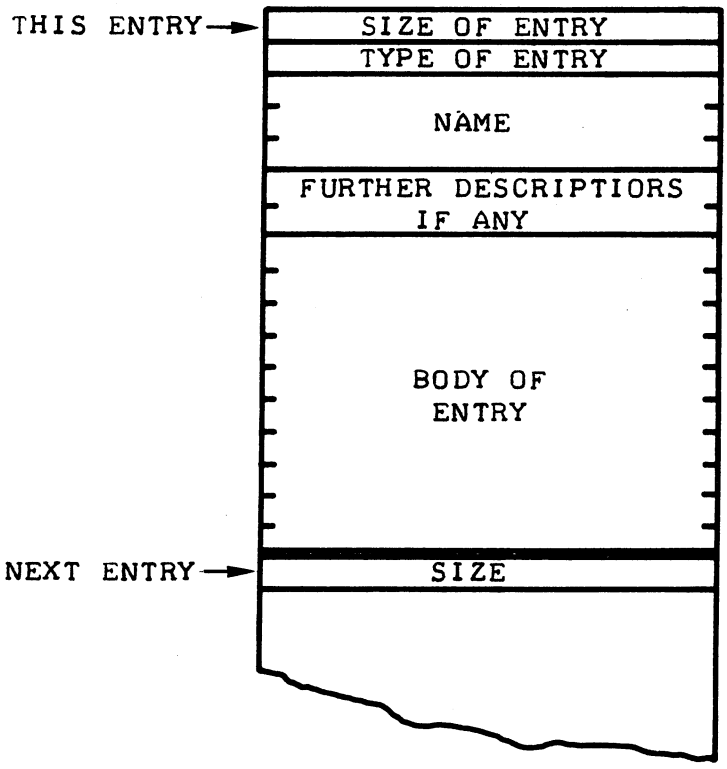
10.7 ARSIZE

This function returns the size of a NODAL array. E.g.

```
$SET STR(ARSIZE(STR)+1) = "NEW ELEMENT"
FOR I = 1,ARSIZE(B); TYPE B(I)
```

11. NODAL element lists

NODAL acts on the accelerator via machine code subroutines, or by data array links to autonomous machine code tasks. These subroutines and data arrays are contained in the NODAL element list. This is a name indexed linked list which can reside in core, on a core-load or both. More than one list can be provided, depending on the computer configuration. The starting address of the list is known to NODAL and so when it encounters a name it can search the list for further information. The structure of the list is shown in the following diagram.



The parts of the entry are used as follows :

a) Size of entry

This has a dual role, telling the interpreter the number of words in the entry as required by some commands, and also providing the link to the next entry since the entries are put end to end.

b) Type of entry

Many different types of entry will be catered for. For example one can cite simple variables, read only variables, arrays, system variable subroutines. A list of NODAL element types is given in appendix 6.

c) Name

This takes three words for the six character name.

d) Further descriptors

These give further information, if required, about the entry. For instance for a machine code subroutine an entry point word and a parameter descriptor word will be required.

e) Body of entry

This contains the data or code. In the case of machine code subroutines the code could be elsewhere as the entry point is given in absolute terms.

11.1 NODAL element types

A summary of element types is given in appendix 6. This section give more information on each type.

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
```

Tape 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 - FORTRAN function call

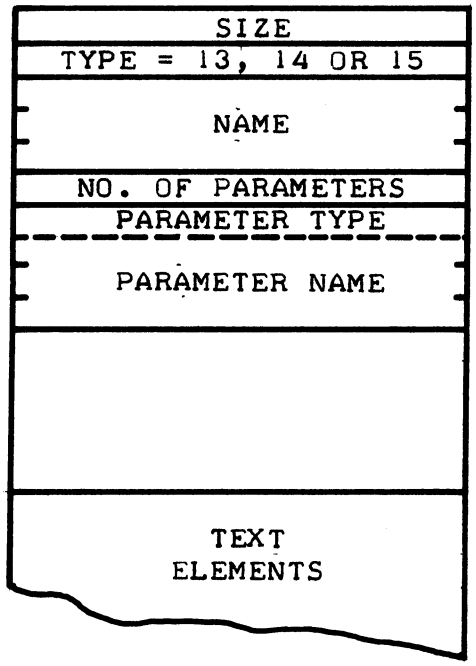
This entry permits NODAL to call a NODAL library function. Two words are used in addition to the header, the number of parameters and the entry point of the routine. The number of parameters can be 1 or 2. The first parameter is held in the TAD and the second pointed to by X. The result is returned in 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 2.11.2 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 chapters 6 and 7.

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

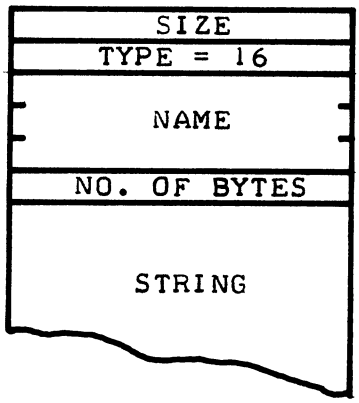


The number of parameters can be -1, 0 or 1-8. -1 is obtained by the DEF-FUN command with no arguments and causes the variables PAR1 and PAR2 to be created as described in chapter 8. 0 means no parameters, and 1 to 8 gives the number of parameters required. For each parameter a descriptor block of 4 words 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.

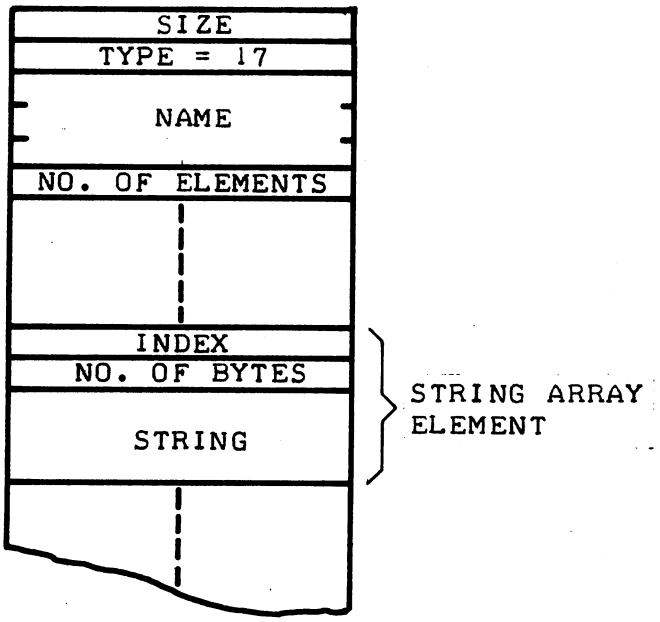
Types 16 and 17 - String variables and arrays

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

STRING VARIABLE



STRING ARRAY



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.

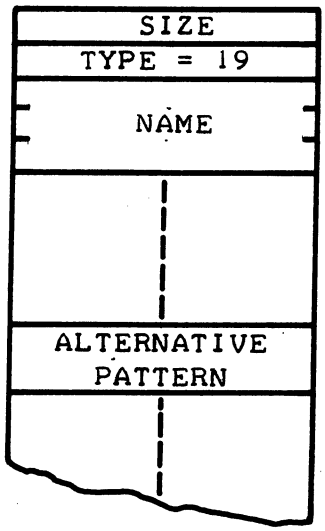
TAD = VALUE return from read function

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

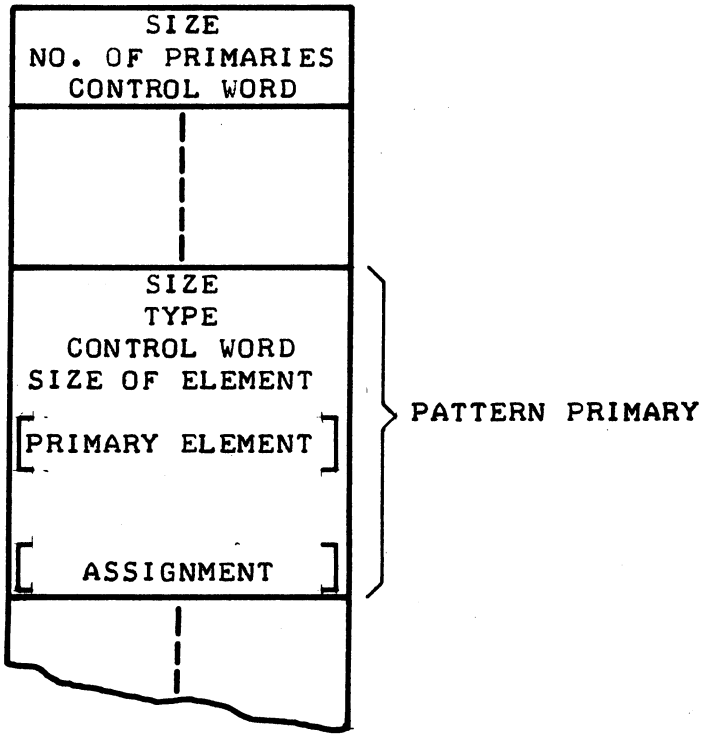
NOTE The fact that T = ADDRESS of element applies to all of types 8, 9, 10, 11, 12, 18, 22, 23 and 24. This can be used to provide core-load switching when the header is resident but the routine is on another core-load.

Type 19 - Pattern variable

The structure of a pattern variable is as follows :



The structure of each alternative pattern is as follows :



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 14 - 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

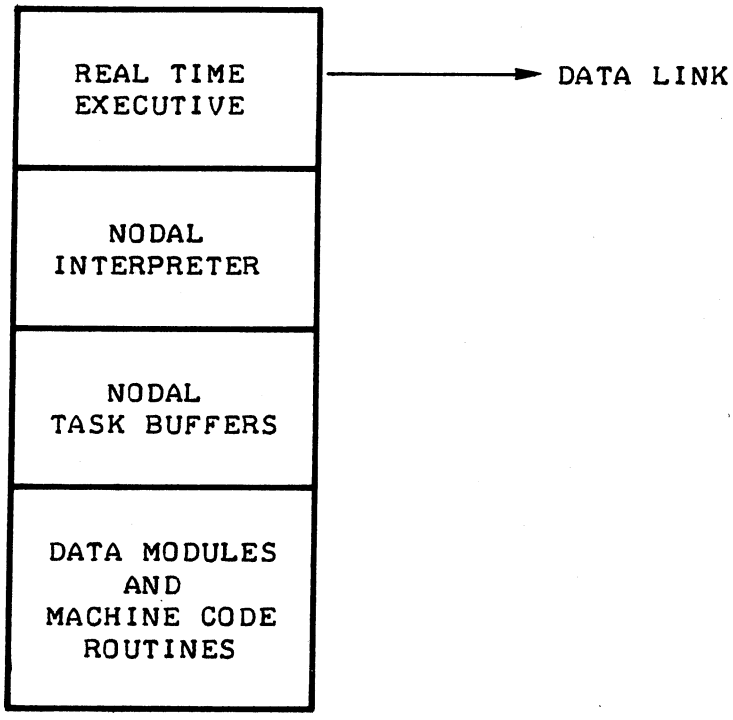
Created temporarily at run-time in a defined function to point to reference variable.

12. Organisation of the NODAL System

This chapter describes how the NODAL system is organised in a particular computer. Four examples are taken, a simple core-only system, a console computer, a general purpose computer, and an experimental area computer. The organisation is described in terms of memory layout and the function of each part of memory.

12.1 Simple core-only system

The core layout of such a system is



The functions of the different parts are as follows.

a) REAL TIME EXECUTIVE

This consists of the basic operating system plus "add-ons". It handles input and output to the computer peripherals, CAMAC and the multiplexer, and the data-link. It also organises tasks in terms of priority and time.

b) NODAL INTERPRETER

This is a re-entrant piece of machine code which causes a NODAL program to be executed. A particular NODAL program is essentially character string data which the interpreter translates into action.

c) NODAL TASK BUFFERS

These hold the NODAL program which is being interpreted, its variables, and the run-time stack which is used by the interpreter. A NODAL task occupies a buffer continuously from initiation to completion time. The actual program in the buffer can change, however, in response to the RUN, LOAD and similar commands. As the NODAL interpreter is re-entrant as many NODAL tasks can be active "simultaneously" as there are task buffers. In this core-only system all buffers must be in core so only three buffers are proposed.

The_Interactive_Buffer is used in normal interactive programming. It is attached to the terminal, and any program which is typed in is stored in this buffer. Also any program RUN from a file is pulled into this buffer for execution. If only one such buffer is available only one terminal can be active at a time.

The_Real-time_Buffer is used for programs sent down from other computers via the EXECUTE command. It is also used by programs HOOKed to interrupts, or SCHEDLed on time. The philosophy of use for this buffer is that many different programs will run in it in the course of one accelerator cycle. Thus programs should be short and able to run directly to completion. In particular the WAIT command should not be used. Normally WAIT's should be done at the interactive level, say in the console computer, and real-time tasks only sent down over the data-links at the right time.

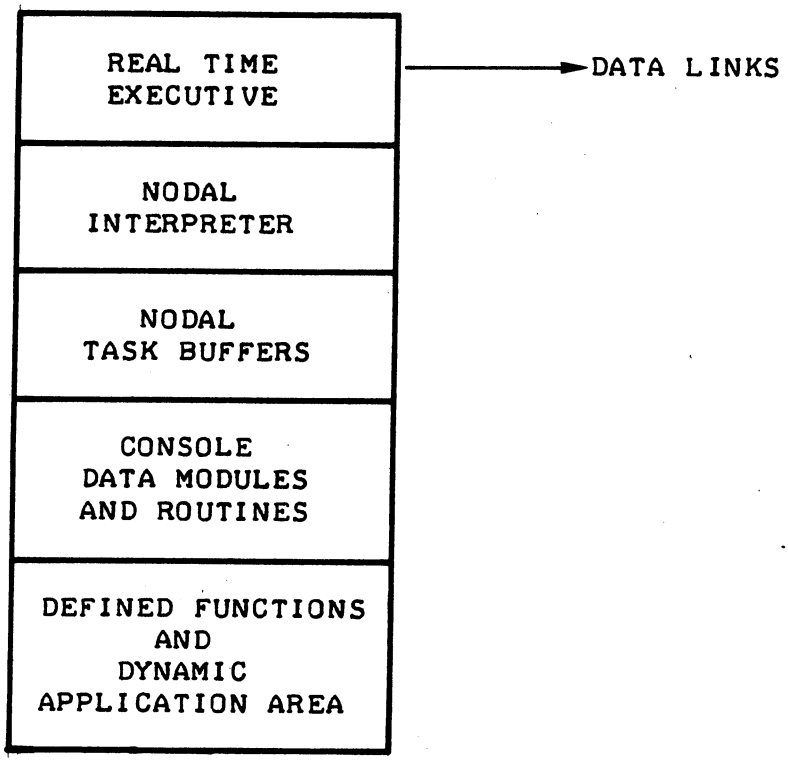
The Immediate Command Buffer is a small buffer meant for handling high priority immediate commands, usually generated remotely using the IMEX command. This will be used operationally, on occasion to directly interrogate some remote data-module, but mostly to interrogate a remote computer's status, or abort some task that has got into a loop.

d) DATA MODULES AND MACHINE CODE ROUTINES

This area contains the special assembly language programmed data-modules and machine code routines to handle the apparatus connected to this particular computer. These are accessed via a NODAL element list as required by the NODAL program currently being interpreted.

12.2 Console computer system

The core layout for a console computer is shown below.



This differs from the simple system described above by the addition of the DYNAMIC APPLICATION AREA. In this area are stored the NODAL defined fuctions, either as created using the DEFINE command, or as loaded from a file. A special system file will contain a large number of system defined functions. When the NODAL interpreter in the console computer encounters a system variable type name which it cannot find in its own lists, it will search this special file for a defined function of that name. If this is found it will be loaded and executed. The function will then remain in the dynamic area for more rapid subsequent use.

A special command will be available in the console computer to load the dynamic area. This is the USE command, for example

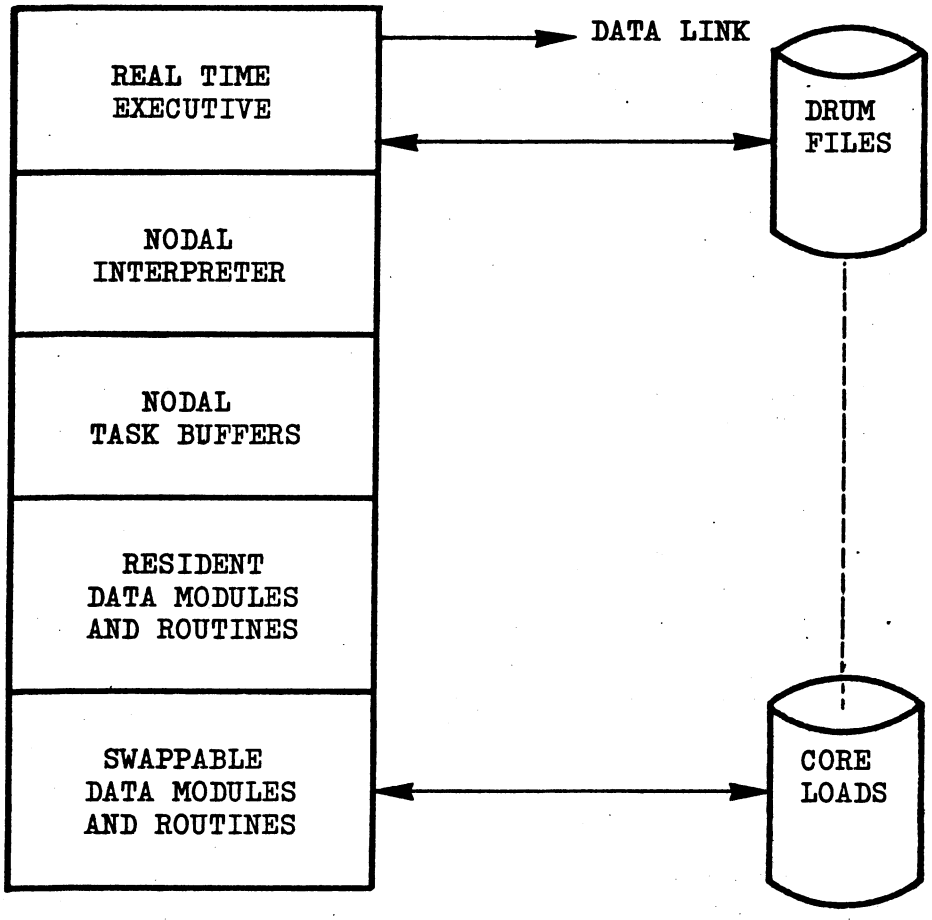
USE MYFILE RF1

will cause the dynamic area to be loaded with the contents of MYFILE and of RF1. These files may contain a selection of pre-defined NODAL functions, or special application routines written in assembly language or even FORTRAN. In this latter case the USE command is acting as a loader.

It must be noted that this dynamic configuration of the console computer can be a slow process. It is intended that the set of routines will be built up for a given application, say at log-in time, then stay more or less constant. The dynamic area can be cleared using the command ZDEF.

12.3 The general purpose computer system

The drum systems are more complicated and are shown schematically below



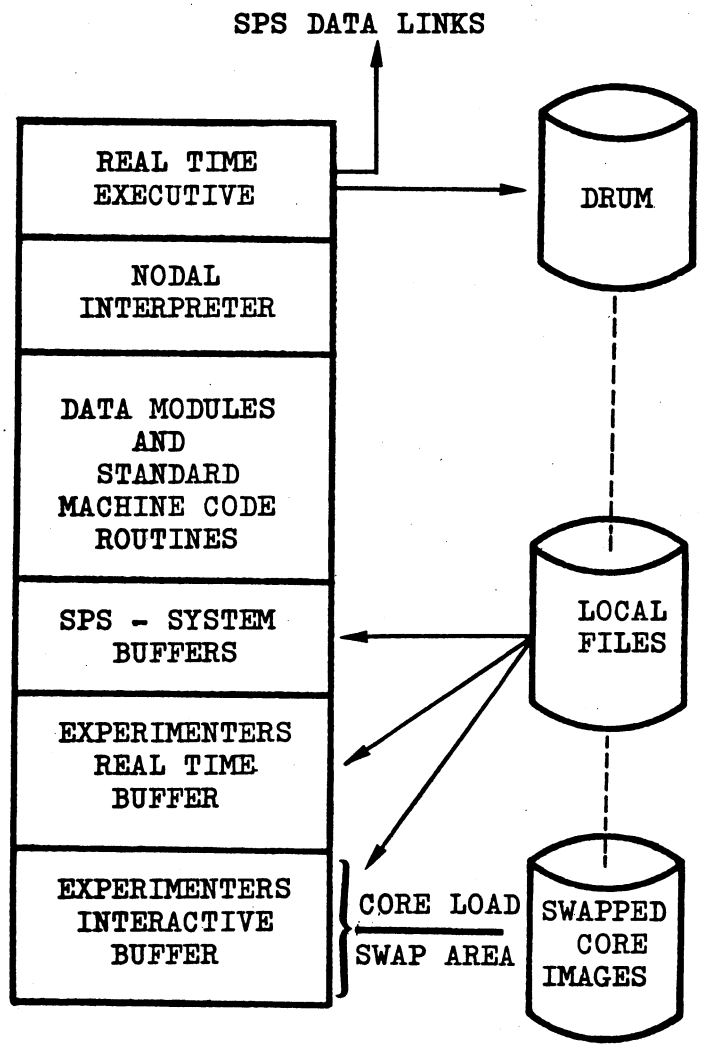
The use of a drum provides two additional facilities. Firstly it provides local files for storage of NODAL programs and data, thus reducing the traffic on the data-link, possibly improving response tiime as compared with the library computer, and providing for stand-alone operation. Secondly it provides a "virtual memory facility". This allows machine code subroutines and tasks to be resident on the drum in "core-image" form, ready to be swapped in whenever required. For this to happen an additional area in core is required, called the core-load area.

The use of core-loads extends the amount of machine code which can be used in a system. A typical system might have a 4K core load area and eight 4K core loads. This allows 32K of machine code programs or subroutines, though only 4K can be resident at any one time. The penalty is the swap-time. This consists of a waiting time for the drum to get into the correct position, then the transfer time. The drum has a rotation time of 20 msec and a transfer rate of 2K per revolution. Thus to read in a 4K core load will take on average 50 msec. A 200 word disk file will take, on average. 12 msec to read. Core-loads are divided into two categories, read-only and write-back. If a program stores data within itself or elsewhere on the core-load, the core-load must be write-back. This increases the swap time to 100 msec average for the 4K core-load.

12.4 Experimental area computer system

The software for the experimental area (EA) computer is different from the usual software in that it it must support several (say up to 10) interactive users. The first of these is the EA control position, probably with the highest priority. Then each experimental team will have a terminal from which it can run programs to interrogate machine status and change its own beam line elements. In addition a direct link between the CAMAC on the EA computer and the CAMAC on the experimenter's computer will be provided. This will allow the experimenter's computer to read either a standard block of data which is updated every cycle by the EA computer, or a special block of data individually prepared by an experimenter's program running in the EA computer.

The following diagram shows the software layout proposed to satisfy the above requirements.



The REAL-TIME MONITOR handles access to the local drum files and SPS DATA-LINK, controls CORE-LOAD switching, priorities etc. The NODAL INTERPRETER causes execution of NODAL programs, usually by generating calls to DATA MODULES and STANDARD MACHINE-CODE routines.

TASK BUFFERS are used to hold NODAL programs which are in the process of being interpreted, together with the run-time stack used by the interpreter. The NODAL interpreter is re-entrant so several task buffers can be used con-currently. Here the TASK BUFFERS are divided into two lots. The first is that reserved for SPS use. In these buffers will be interpreted immediate commands and programs sent down from other computers in the SPS system, together with the closed loop and surveillance programs scheduled routinely. The other lot of task buffers consists of the EXPERIMENTER'S REAL-TIME BUFFER and the INTERACTIVE BUFFERS. The use of these buffers is described further below. The INTERACTIVE BUFFERS all occupy the same locations in core memory, but have a unique location on drum. The are swapped in and out as necessary by the system, the swap time for a 2K buffer being about 60 milliseconds.

INTERACTIVE USE

From his terminal any user can obtain information about his beam line using the standard features of NODAL and the DATA MODULES contained in the EA computer. Typically he would run a program which would print out a table of currents, positions, scaler values etc. He can also control his beam-line elements, either directly using immediate commands, or via NODAL programs for scans, etc. Control of beam-line equipment goes through the DATA MODULES and is vetted by a password system so that each terminal will only have limited access.

INTER-COMPUTER USE

The experimenter's CAMAC will be linked to the EA CAMAC by a link module. By writing a word into this module the experimenter can create an interrupt in the the EA computer and get some response. Depending on the word written, the response can be of two forms. Firstly a standard block of EA data can be sent back over the link module. Secondly a specific user written NODAL program can be activated. In this latter case the NODAL call

TRANSF(A)

will send the contents of the array A (integer array of some maximum size, say 50 words) back over the link. The NODAL program which does this can be written in the normal way and stored on a local drum file. It is then

connected to the interrupt from the experimenter's computer by the NODAL call, eg.

```
HOOK(FILE,3,2)
```

where FILE is where the program was stored, 3 is the user number, and 2 is logical number of his request as will be determined from the number he first writes into the link module to create the interrupt. Some utility commands are

```
DISCON(FILE)
```

which disconnects the HOOK set up above, and

```
ZTRANS
```

which clears and resets the link status at the EA end.

Command Summary

NODAL commands can be abreviated provided there is no ambiguity. Commands, whether abbreviated or not, must be followed by a spàce or some non-alphabetic character. The first three letters of a command will always be unambiguous, though sometimes fewer will be required.

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.

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

ERASE	ERASE 1.1 2 A B	Erase elements from local program storage.
	ERASE	Erase all variables
	ERASE ALL	Erase all user input.
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;(C) FOR A = X,Z;(C)	(C) is any sequence of commands stretching to end of line (including more FOR commands). (C) is executed with values of A ranging from X to Z by increments of Y. Y if assumed 1 if not present.
GOTO	GOTO 3.4 GOTO X	Transfers control to line 3.4 or line given by expression X rounded to 2 places after point
IF	IF (X)Ln,Ln,Ln	Transfers control to first, second, or third line number according to wether X is negative, zero or positive.
	IF(X) Ln,Ln; IF(X) Ln;	Transfers control according to conditions satisfied, otherwise continues with commands after.
	IF X>Y; --- IF Z<=Y; --	Logical IF. DO rest of line if condition satisfied Conditions are : = < > <= >= <>
IMEX	IM(5) TYPE BCT(3)	Send command to computer 5 for immediate execution. Result typed out on terminal where IMEX command given
LIST	LIST LIST 1.1 LIST 1 LIST 1.1 2 3	NODAL types out entire program. Types out line 1.1 Types out all group 1. Types line 1.1, group 2 and 3
LOAD	LOAD FILE	Loads contents of FILE into task buffer. FILE can contain data or lines. Any corressponding elements in task buffer are deleted and replaced by corresponding ones from FILE

OLD	OLD FILE	Loads program from "FILE".
OPEN	OPEN FUNCTN	Copy text of defined function into task buffer and delete function
OVERLAY	OVE FILE X,Y	Run program stored on FILE as an overlay in the task buffer. Program disappears afterwards. ARG(1) and ARG(2) are set to X and Y
QUIT	QUIT	Returns control to Monitor
REMIT	REM A B	Sends data items A and B back to program which caused, via EXECUTE command, REMIT to occur
RETURN	RETURN	Return from DO group.
RUN	RUN RUN FILENAME RUN [X] FILENAME	Run program just typed in Run program stored on filename Run starting from line X
SAVE	SAVE FILE SAVE FILE 2 A B	Save program on "FILE". Save specified material on "FILE".
SET	SET A = X	Sets variable A to value of expression X.
TYPE	TYPE A+B-C	Evaluates expression and types out result.
	TYPE [LIST]	Types out [List] were [List] can contain expressions, strings, and control sequences, such as ! for carriage return line feed # for carriage return %X for format change to X &X for X spaces
USE	USE FILE	Append "FILE" to console computer system variable list
VALUE	VAL X	Exit from NODAL function with X as value.

WAIT	WAIT (X)	Wait for data to be remitted from computer X
	WAIT-TIME X	Wait for X seconds
	WAIT-CYCLE 4	Wait for event 4 in accelerator cycle
WHILE	WHILE A<2; DO 3	Does rest of line after ";" while condition is satisfied
%	% THIS IS A COMENT	Causes rest of line to be treated as a comment.
?ON ?OFF		Switches trace feature ON or OFF.
$SET	$SET A="QWER"	Sets string variable A to contain QWER. Creates A if it does not already exist
	$SET A=INPUT(X):X	Set A to result of function INPUT. Go to line represented by X if function fails
$ASK	$ASK"STRING IS"A	Reads string from terminal (terminated by carriage return) stores string in A, creates A if non-existent
$IF	$IF(A-B)Ln,Ln,Ln	compares strings A and B lexically. Branches as in arithmetic IF
$DO	$DO STR1	Does command stored in STR1
$VALUE	$VALUE "XYZ"	Exit from NODAL function with string "XYZ" as value
$PATTERN	$PAT P1 = "XYZ"	Create the pattern P1, here the string "XYZ"
$MATCH	$M STR1 P1 : X	Match the string STR1 with the pattern P1. Goto line given by X if match fails

LINE EDIT COMMANDS

(CTRL)A Backspace one character (types ↑)
(CTRL)C Copy one character from old line to new line
(CTRL)D Copy rest of old line and terminate edit
(CTRL)E Change insert/replace mode (types open arrows)
(CTRL)F Copy rest of old line without typing, and terminate

(CTRL)H Copy rest of old line without terminating edit
(CTRL)I Space to next tab stop
(CTRL)L Terminate edit
(CTRL)M Terminate edit with carriage return
(CTRL)OC Copy old line up to but not including character C

(CTRL)PC Skip characters in old line up to character C
(CTRL)Q Backspace to the beginning of the new and old lines
(CTRL)R Retype fast
(CTRL)S Skip one character in old line (types %)
(CTRL)T Retype aligned

(CTRL)U Copy up to the next tab stop
(CTRL)VC Take character C literally
(CTRL)W Backspace one word (types \)
(CTRL)XC Skip characters in old line up to and including C
(CTRL)Y Append rest of old line to new line and edit result
(CTRL)ZC Copy old line up to and including character C

```
 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   CAM,    FUSIZ; 10; #CA; #MA; #C$-##$; 0; 21042; CAM1
 6
 7   PARST = 23-200
 8   VALUE = PARST
 9   FLAG  = PARST+3
10   CRATE = PARST+4
11   MODUL = PARST+5
12   SUBAD = PARST+6
13   FUNCT = PARST+7
14   SAVEL = PARST+10
15   NAF   = PARST+11
16   CRIO  = PARST+12
17
18   CAM1,      COPY DX SL; STX SAVEL,B
19              LDA CRATE,B; JAN CAMER; AAA -20; JAP CAMER
20              AAA 20; SHA 6; STA CRIO,B
21              LDA MODUL,B; JAN CAMER; AAA -40; JAP CAMER
22              AAA 40; SHA 11; STA NAF,B
23              LDA SUBAD,B;JAN CAMER; AAA -20; JAP CAMER
24              AAA 20; SHA 5; ORA NAF,B; STA NAF,B
25              LDA FUNCT,B; JAN CAMER; AAA -40; JAP CAMER
26              AAA 40; ORA NAF,B; STA NAF,B
27              LDA FUNCT,B
28              AAA -10; JAN CAMR
29              AAA -10; JAN CAMC
30              AAA -10; JAN CAMW
31   CAMC,      LDA FALG,B; JAN CAMER
32              LDX CRIO,B; LDT X2037; RORA DT SX
33              LDA NAF,B; EXR ST
34              LDT X2070; RORA DT SX; EXR ST
35   CAM2       NLZ 20; STF VALUE,B
36   CAM2A,     SAA 0
37   CAM3,      STA FLAG,B; JMP I SAVEL,B
38   CAMER,     SAA 24; JMP CAM3
39   CAMR,      LDA FLAG,B; JAN CAMER
40              LDA X2037; LDT CRIO,B; RORA DT SA
41              LDA NAF,B; EXR ST; JMP CAM2
42   CAMW,      LDA FLAG,B; JAP CAMER; LDX CRIO,B
43              LDF VALUE,B; JPL I (NFIX
44              LDT X2021; RORA DT SX; EXR ST
45              LDA NAF,B; LDT X2037; RORA DT SX
46              EXR ST; JMP CAM2A
47   X2037,     IOX 2037
48   X2070      IOX 2070
49   X2021,     IOX 2021
50              )FILL
51   FUSIZ=*-CAM
52              )PCL CAM
```

EXAMPLE DATA-MODULE SETUP

```
 2    % EXAMPLE DATA-MODULE SETUP
 3    % SHOWS THREE SYSTEM VARIABLES
 4    % MBBH (ONE OFF), MBBV (ONE OFF), MBBQ (TWO OFF)
 5    % EQUIPMENT TYPE 2  (0,1,2,3, -- 63)
 6    % ONLY ONE PROPERTY, SAY #CUR
 7    % SETS VALUE IN TABLE ON WRITE
 8    % TAKES VALUE FROM TABLE ON READ
 9
10           7; 7; #MB; #BH; 0; 4144; 4144
11           7; 7; #MB; #BV; 0; 4145; 4145
12           7; 7; #MB; #BQ; 0; 4146; 4147
13           -1
14
15    SVTAB, 0
16           0
17           EQT2
18
19    % DATA-MODULE FOR EQUIPMENT TYPE 2
20    % ONLY ONE PROPERTY, #CUR = 16659 (040423)
21    % HANDLES EQUIPMENT NUMBERS 4144 TO 4147
22    % SETS VALUE IN, AND READS VALUE FROM, DATA TABLE DTB2
23
24    EQT2=*
25
26           SERNO=-162
27           RETAD=-161
28
29           AAB -20; COPY DX SL; STX RETAD,B
30           COPY DX SA; SWAP DX SB
31           LDA I 2,B; SAD ZIN SHR 12; AAA -2; JAF EQNER
32           SAD 12; AAA -144; JAN EQNER; STA SERNO,X
33           AAA -4; JAP EQNER
34           LDA I 3,B; JAZ PRPCR; SUB (040423; JAF PRPER
35    PRPCR, LDA I 1,B; JAP READ
36           LDF I 0,B; JPL I (NFIX
37           LDT SERNO,X; SWAP DX ST; STA I DTABL,X
38           SWAP DX ST; JMP EOUT
39    READ,  LDT SERNO,X; SWAP DX ST; LDA I DTABL,X
40           SWAP DX ST; NLZ 20; STF I 0,B
41    EOUT,  STZ I 1,B
42    EQEX,  COPY DB SX; LDX RETAD,B; AAB 20; JMP 0,X
43    EQNER, SAA ERR35; JMP EQF
44    PRPER, SAA ERR36
45    EQF,   STA I 1,B; JMP EQEX
46    DTABL, DTB2
47           )FILL
48           )PCL EQT2
49
50    DTB2,  0;0;0;0
51           )LINE
```

NODAL ERROR NUMBERS

```
ERR1=1                  % ILLEGAL LINE NUMBER
ERR2=2                  % ILLEGAL FORMAT SPECIFICATION
ERR3=3                  % ILLEGAL ARITHMETIC EXPRESSION
ERR4=4                  % AMBIGUOUS COMMAND
ERR5=5                  % ILLEGAL DELIMITER
ERR6=6                  % ATTEMPT TO DIVIDE BY ZERO
ERR7=7                  % WORKING AREA FULL
ERR8=10                 % NONEXISTENT NAME
ERR9=11                 % ILLEGAL TYPE
ERR10=12                % ILLEGAL SET COMMAND
ERR11=13                % COMMAND NOT PROPERLY TERMINATED
ERR12=14                % ERROR IN FOR COMMAND
ERR13=15                % NONEXISTENT LINE ADDRESSED
ERR14=16                % ILLEGAL SHUFFLE IN DEFINED FUNCTION
ERR15=17                % ERROR IN IF COMMAND
ERR16=20                % ESCAPE TYPED
ERR17=21                % ILLEGAL EDIT COMMAND
ERR18=22                % ILLEGAL ASK COMMAND
ERR19=23                % ERROR IN ERASE COMMAND
ERR20=24                % ARGUMENT LIST ERROR
ERR21=25                % FILE ERROR
ERR22=26                % ERROR IN SAVE COMMAND
ERR23=27                % ARRAY DIMENSION ERROR
ERR24=30                % SQUARE ROOT OF NEGATIVE NUMBER
ERR25=31                % ILLEGAL ARCTANGENT ARGUMENTS
ERR26=32                % SINE ARGUMENT TOO BIG
ERR27=33                % COSINE ARGUMENT TOO BIG
ERR28=34                % POWER ERROR [NEGATIVE ARGUMENT?]
ERR29=35                % POWER UNDERFLOW
ERR30=36                % EXPONENTIAL ARGUMENT TOO BIG
ERR31=37                % LOGARITHM ARGUMENT <=0
ERR32=40                % DEVICE NOT CONNECTED
ERR33=41                % UNAUTHORISED ACTION
ERR34=42                % HARDWARE ERROR
ERR35=43                % ILLEGAL EQUIPMENT NUMBER
ERR36=44                % ILLEGAL PROPERTY
ERR37=45                % VALUE OUT OF RANGE
ERR38=46                % NOT IMPLEMENTED YET
ERR39=47                % NO SUCH COMPUTER
ERR40=50                % RESULT STRING FILLED

" 88DEF+88STR
DERR1=51                % DEFINED FUNCTION AREA FULL
DERR2=DERR1+1           % SYNTAX ERROR IN DEFINE COMMAND
```

NODAL ERROR NUMBERS CONTINUED

```
"88STR
SERR1=53              % ILLEGAL STRING SET COMMAND
SERR2=SERR1+1         % STRING FUNCTION FAILURE
SERR3=SERR2+1         % ILLEGAL CONCATENATION
SERR4=SERR3+1         % ERROR IN $IF COMMAND
SERR5=SERR4+1         % ERROR IN $ASK COMMAND
SERR6=SERR5+1         % STRING EXPECTED

"88PAT
PERR1=61              % PATTERN TOO BIG
PERR2=PERR1+1         % BAD PATTERN MATCH
PERR3=PERR2+1         % BAD PATTERN
PERR4=PERR3+1         % BAD PATTERN ASSIGNEMENT
"
```

NODAL ENTRY TYPES

		handled by
1.	LINE ENTRY	-
2.	SIMPLE VARIABLE	DIRECT
3.	READ ONLY VARIABLE	DIRECT
4.	NUMERIC ARRAY	ARRAY
5.	READ ONLY ARRAY	ARRAY
6.	FORTRAN FUNCTION	GFUN
7.	SYSTEM VARIABLE	SYSVR
8.	READ/WRITE GENERAL PURPOSE MODULE	ASSFN
9.	WRITE ONLY GENERAL PURPOSE MODULE	ASSFN
10.	READ ONLY GENERAL PURPOSE MODULE	ASSFN
11.	CALL, NODAL ERROR RET G.P. MODULE	ASSFN
12.	CALL NO ERROR RET G.P. MODULE	ASSFN
13.	READ/WRITE NODAL FUNCTION	NODFN
14.	NODAL CALL SUBROUTINE	NODFN
15.	NODAL READ/WRITE STRING FUNCTION	NODFN
16.	STRING VARIABLE	DIRECT
17.	STRING ARRAY	DIRECT
18.	FREE MACHINE CODE ROUTINE	DIRECT
19.	PATTERN VARIABLE	DIRECT
20.		
21.	PATTERN FUNCTION	DIRECT
22.	READ/WRITE STRING FUNCTION	DIRECT
23.	WRITE ONLY STRING FUNCTION	DIRECT
24.	READ ONLY STRING FUNCTION	DIRECT
25.	REFERENCE POINTER	GNAM