

H-CODE MANUAL

by  
D. C. Hogg

---

SUMMARY:- This manual is a description of the facilities available in the programming system in use on the Brush 803 computer. It will be supplemented from time to time by additional sections. The manual does not attempt to teach programming; it is a sequel to RDR/100, which introduces H-Code.

---

September, 1963

---

BRUSH ELECTRICAL ENGINEERING CO., LTD.  
A MEMBER OF THE HAWKER SIDDELEY GROUP  
LOUGHBOROUGH, ENGLAND

---

## Contents

<b>1</b>	<b><u>A PROGRAMME</u></b>	<b>1</b>
<b>2</b>	<b><u>"SETTING" THE MACHINE</u></b>	<b>1</b>
2.1	SETTING PROCEDURES . . . . .	2
2.2	SETTING OF VARIABLES . . . . .	2
2.3	SETV, SETS . . . . .	2
<b>3</b>	<b><u>LOCAL VARIABLES</u></b>	<b>3</b>
<b>4</b>	<b><u>LIST (DATA)</u></b>	<b>4</b>
4.1	DATA . . . . .	5
<b>5</b>	<b><u>THE BODY OF A PROCEDURE OBEYING A PROCEDURE</u></b>	<b>6</b>
5.1	THE BODY OF A PROCEDURE . . . . .	6
5.2	OBEYING PROCEDURE . . . . .	6
<b>6</b>	<b><u>BINARY</u></b>	<b>6</b>
<b>7</b>	<b><u>SUNDRY ORGANISATIONAL INSTRUCTIONS</u></b>	<b>6</b>
<b>8</b>	<b><u>ERRORS IN PROCEDURE</u></b>	<b>7</b>
<b>9</b>	<b><u>SUMMARY OF INITIAL (OR ORGANISATIONAL) INSTRUCTIONS</u></b>	<b>7</b>
<b>10</b>	<b><u>FLOATING POINT FORM OF NUMBERS</u></b>	<b>7</b>
10.1	FLOATING POINT FORM . . . . .	7
10.2	FORM OF NUMBERS . . . . .	8
<b>11</b>	<b><u>MACHINE CODE BLOCKS</u></b>	<b>8</b>
<b>12</b>	<b><u>USE OF PROCEDURES</u></b>	<b>9</b>
<b>13</b>	<b><u>STRUCTURE OF A PROCEDURE</u></b>	<b>10</b>
<b>14</b>	<b><u>LOOP AND DO INSTRUCTIONS</u></b>	<b>10</b>
<b>15</b>	<b><u>ON STATEMENT</u></b>	<b>12</b>
<b>16</b>	<b><u>SUFFICES</u></b>	<b>12</b>
16.1	SIMPLE SUFFICES . . . . .	12
16.2	COMPLEX SUFFICES . . . . .	13
<b>17</b>	<b><u>JUMP SUFFICES, UNCONDITIONAL JUMPS</u></b>	<b>13</b>
17.1	JUMP SUFFICES . . . . .	13
17.2	UNCONDITIONAL JUMPS . . . . .	13
<b>18</b>	<b><u>CONDITIONAL STATEMENTS</u></b>	<b>14</b>
18.1	GENERAL CONDITIONAL STATEMENT . . . . .	14
<b>19</b>	<b><u>BRANCH</u></b>	<b>16</b>
<b>20</b>	<b><u>READ STATEMENTS</u></b>	<b>16</b>
<b>21</b>	<b><u>PRINT STATEMENTS</u></b>	<b>18</b>
<b>22</b>	<b><u>REFERENCE NUMBERS</u></b>	<b>18</b>
<b>23</b>	<b><u>BRACKETS OR DEPTH OF EXPRESSIONS</u></b>	<b>19</b>
<b>24</b>	<b><u>CONSTANT SEARCHING</u></b>	<b>19</b>
<b>25</b>	<b><u>CHECK</u></b>	<b>20</b>

---

26	<u>TITLE AND OPTIONAL TITLE</u>	20
26.1	TITLE . . . . .	20
26.2	OPTIONAL TITLE . . . . .	20
27	<u>TRACE</u>	21
28	<u>ARITHMETIC STATEMENT</u>	21
29	<u>ARITHMETIC OPERATIONS</u>	21
30	<u>ALGEBRAIC FUNCTIONS</u>	22
31	<u><math>\alpha</math>-<math>\theta</math> STORES</u>	24
32	<u>BAR A <math>\bar{A}</math></u>	25
33	<u>SUNDRY INSTRUCTIONS IN BODY OF PROCEDURE</u>	25
34	<u>TELEPRINTER CHARACTER: LAYOUT OF PROGRAMME ON TELEPRINTER SHEETS</u>	26
35	<u>WORDS</u>	26
36	<u>SUMMARY OF FACILITIES WITHIN PROCEDURE</u>	26
37	<u>USE OF COMPUTER KEYBOARD</u>	27
38	<u>ERROR INDICATION</u>	29

#### List of Figures

1	The use of a subroutine . . . . .	10
---	-----------------------------------	----

#### List of Tables

1	Effect of characters occurring under LIST . . . . .	5
2	Summary of initial (organisational) instructions . . . . .	7
3	Effect of characters met in obeying READ instruction . . . . .	17
4	Summary of facilities within procedure . . . . .	27
5	Use of Computer Keyboard . . . . .	28

#### Notation

$\xi$	any arithmetic expression
$\xi_v$	any arithmetic expression with final mode floating point
$\xi_s$	any arithmetic expression with final mode fixed point
ABC	floating point variables ) unless otherwise
ILMN	fixed point variables ) specified or implied
	Other notation explained in context

---

#### DISTRIBUTION

Dr. L.R. Blake  
Dr. D.A. Jones  
Central Reference Library

Research Division Library  
and as required

---

## H-CODE MANUAL

by D.C. Hogg

### 1. A PROGRAMME

An H-Code programme (in the sense of a whole job) consists of a number of procedures (or subroutines) which perform arithmetic and logic on fixed and floating point variables represented by letters of the alphabet.

There are two distinct parts of a programme (as opposed to a procedure).

- (i) The body of a procedure — that is the mnemonic algebraic and logical instructions which are obeyed dynamically performing the particular job for which the programme has been composed.
- (ii) The organisational instructions which occur outside the body of the procedures and are concerned with the organisation and setting of the machine ready to receive the procedures.

Purely organisational instructions can occur inside procedures but these are of a different type of those under sect. (ii) being more concerned with the particular procedure than with overall setting.

There are a different set of rules for (i) and (ii) and anything which is true for (i) is not necessarily true for (ii) and vice-versa,

### 2. "SETTING" THE MACHINE

In a statement such as "setting the machine" it should be understood that it refers not to the machine as such which is unchangeable in form, but to the state of the store of the machine and more particularly to the state of the H-Code translator which in normal circumstances is permanently in (the store of) the machine. Once a programme is in the machine any further procedures or data etc are understood to be part of that programme as the machine, so to-speak, is "conditioned" for that particular programme. Therefore to get out of this state of affairs all programmes should be headed by the instruction

:SET

which cancels all reference to previous programmes held in the machine leaving the whole of the available space free for the new programme.

The machine is now set up for the particular programme it is to receive. It has to be prepared for the procedures of the programme, the variables and integers. These settings are introduced by

:SETR :SETV :SETS

In fact when a programme has one of these three settings the SET is not needed as the first

:SETR :SETV :SETS

implies

:SET

itself. Therefore in the great majority of programmes the instruction

:SET

need never be used. Its practical use is limited to introducing machine order programmes and special programmes which can occur under

:LIST

:SETR :SETV :SETS

can occur in any order and more than once but when the first "non-SET" organisational instruction

:DEFINE :LIST :LV :START :Z ...

is encountered the setting instructions are understood to have finished and the machine is finally set up for that programme. Any subsequent

:SETR :SETV :SETS

will cancel the programme.

---

## 2.1. SETTING PROCEDURES

Procedures are identified by words of up to 4 letters. The programmer can choose any word with which to identify a procedure as long as it does not clash with any standard word which occurs in the body of the procedure. (see WORDS). A word of 4 letters can have extra letters added to give it meaning if necessary, but these extra letters have no significance as far as the translator is concerned. For example, we can not name two routines

```
DESIGN A
DESIGN B
```

as they are the same in the first 4 letters.

In order that the machine is ready to input procedures as they are given to it and so that a procedure may refer to a procedure which is not yet in the machine, we have to announce as part of the setting instructions the names of all the procedures that will go to make up the whole programme.

Reference numbers (q.v) are the basic framework on which a procedure is built and referred to in the machine. In order that enough space be left for this framework each procedure name has, as a suffix the maximum reference number which will occur in that procedure.

A typical setting procedure would be:

```
:SETR(ABC13, :XYZ(2), LMN)
```

Notes (i) CR,LF,sp<sup>1</sup> are ignored before the words.

(ii) If there are no reference numbers no suffix is needed.

(iii) Each word can be preceded by : but this is not necessary.

## 2.2. SETTING OF VARIABLES

When a letter representing a variable (fixed or floating point) is encountered in a procedure reference is made to the corresponding entry in a table which gives information on the variable represented by that letter of the alphabet (including special variable  $\alpha$ ) e.g. whether the variable is set, its mode (i.e. fixed - or floating - point) and its absolute location (i.e. the storage location in the machine which has been allocated to  $A_0$ ).

There are in fact two such tables:- that are called the local alphabet and the permanent alphabet. The local alphabet is the table which is referred to in a procedure as explained above. A copy is kept of the initial settings in the permanent alphabet so that after the local alphabet is altered by the use of local variables (q.v.) the local alphabet can be restored to its original state. This and other uses are explained in more detail below.

## 2.3. SETV, SETS

Any of the alphabet can be used to represent a variable. The programmer must decide which letters shall represent floating-point numbers and which shall represent integer variables. This does not restrict to the use of 27 variables as a letter can have a numerical suffix e.g.  $A_3$ ,  $N_{20}$ ,  $\alpha_{10}$  giving us any range of variables we need.

Before the procedures of a programme can be input the machine must reserve space in the store for the variables which are to be used. This is done using (organisational) instructions such as:

```
:SETV(A23, N5, B)
:SETS(I10, J(3), K, L, M2)
```

Those letters under SETV will represent floating-point variables, those under SETS integer variables, Letters can not occur under both SETV and SETS. The number is the highest suffix which the letter will need throughout the programme. If a letter occurs under SETV or SETS more than once then the greatest suffix is taken.

The word SETV or SETS can occur more than once so that

```
:SETV(A3, B10, C20)
:SETR(ABC3, XYZ)
:SETV(B10, A20, C30)
```

Will be equivalent to:-

```
:SETR(ABC3, XYZ)
:SETV(A20, B10, C30)
```

SET (or the first SETV, SETS or SETR which implies SET) automatically sets the orders

```
:SETV(09)
:SETS(09)
```

---

<sup>1</sup>sp<sup>2</sup> = 2 consecutive spaces

This means that  $\alpha, \theta$  can only be SET as variable and integer respectively. In fact  $\alpha, \theta$  should not (by convention) be used for any other purpose other than in  $\alpha - \theta$  stores (q.v.) i.e. In transferring data from one procedure to another.

When the first non-SET instruction is encountered copies are made in both the local and permanent alphabets of SET-tings as specified by the SETV and SETS Instructions.

#### Notes.

- (i) No check is made either in translation or in running that the "SET" value of the suffix to a letter has been exceeded, i.e. a variable has "gone too high". In fact, it is true to say that 90% of programme errors not detected by the translator are due to this fault.
- (ii) Throughout this manual "variables" will mean floating-point variables and "integers" fixed—point variables - where there is no ambiguity.
- (iii) A can be used instead of  $A_0$ .
- (iv) CR, LF spsp, are ignored before the letters on the RHS of the SETV, SETS instructions.

### 3. LOCAL VARIABLES

It is very convenient, for example, in writing a procedure (Particularly of a mathematical or other independent nature) which is to be used as a subroutine (q.v.) by other procedures to be able to use any letters to represent the variables and integers in the subroutine without reference to the meanings of the letters as SET in the master programme.

To do this we use what are called local variables (and integers). When these variables are used only the local alphabet is affected. Setting of local variables is done with orders such as

```
:LV (A30, B10, X)
:LS (N2, L13)
:LR
```

Those letters under LV will represent floating point variables and those under LS integers. When this instruction is read the machine will, for example, reserve 31 locations for  $A_0 \rightarrow A_{30}$  etc., and make a note in the local alphabet of the new definition of A. As settings merely cancel any previous setting of a particular letter then

```
:LV (A30, A3)
```

will reserve locations for  $A_{30}$  then for  $A_3$ , the  $A_3$ , superseding the  $A_{30}$  The maximum suffix is not taken as it is in SETV, SETS.

Inside LV or LS we can have a setting such as

```
A →B30
```

This means that A in the local alphabet, takes the properties equivalent to B30 in the permanent alphabet. In effect this means that whenever A is met the machine reads it as if it were  $B_{30}$ ,  $A_1$  as  $B_{31}$  etc.

~~Whether this transfer occurs under LV or LS is irrelevant as the mode of the letter A for example, takes its mode from B. The A can not have a suffix.~~

#### Hand written notes:

:L(A →B30)	Will have mode of B30
:LS(A →B30)	Will be INTEGER regardless of mode of B30
:LV(A →B30)	Will be FLOATING-POINT

A setting such as

```
A →A
```

will replace the local meaning of A by the permanent definition of A.

Inside a LV or LS we can also have a setting such as

```
A : 3050
```

This means that  $A_0$  in the local alphabet is to be the absolute location 3050 in the machine. The mode is determined by LV or LS. This use of absolute locations should only be used with great care as a knowledge of the H-Code translator is needed. It can be used to overwrite parts of the translator which are not needed by that particular programme in order to get more working space. No indication is made in this kind of setting as to the maximum suffix of the letter.

The setting

A:0

irrespective of the mode of A, cancels the setting of A in the local alphabet. Thereafter A, for example, will be rejected. This means that a variable or integer can not be set to absolute machine location 0.

These various settings can all occur under the same LV or LS e.g.

:LV(A30,B →C21,X:2000,W:0)

Associated with the setting and manipulation of local and permanent alphabets are the instructions:—

:PL  
:PR

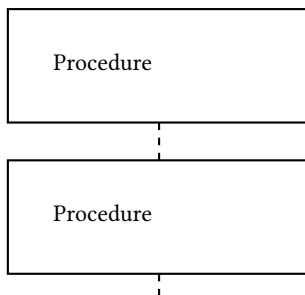
which transfers the permanent alphabet to the local alphabet en masse, and

:LP

which transfers the local alphabet to the permanent alphabet en masse. Individual items can not be transferred from the local alphabet to the permanent alphabet.

A procedure or group of procedures which have their own local variables should in general (although not necessarily) be followed by PL so that the variables then take on their permanent definition. e.g.

:LV(A20,B10)  
:LS(X,N2)



:PL

#### 4. LIST (DATA)

After reading the initial instruction LIST the machine is in a position to accept.

- (i) Lists of variables in fixed-or floating-point form or blocks of machine code introduced by the address at which they are to be stored.
- (ii) Blocks of H-Code programme which is translated, obeyed "there and then", and then over written.
- (i) It is very convenient to be able to set lists of constants (which are unaltered throughout the running of the programme) as the programme is translated rather than as the programme is obeyed dynamically. For example:-

:LIST

A  
+265 300.42  
-22.6  
C10=0 1 10  
5 -6 2  
@=3.6  
N<

} *Machine – codeBlock*

)

→ 3000

<23A:04A2)  
\*

A number is not accepted unless the address is set. The address is the variable to which the first number is to be sent. As each number is read in the address is updated by 1. In the above example A = 265, A1=300.42, A2 = -22.6. If the variable is an integer, for example, only integers can be listed under that address, < ... ) contains a machine code block which is stored beginning at the current address. see MACHINE CODE BLOCKS.

Numbers and machine code orders can be mixed under the same initial address,  $\rightarrow$  3000 sets the absolute location 3000 as the initial address. Only integers or machine code blocks can be under this address.

- (ii) Under LIST between brackets ( ) we can write orders as if it were a procedure. All can be used other than reference numbers. The order:

```
:GOTO(26)
```

would refer to absolute location 26.

This facility is useful for setting up constant data in translation which is too complicated to set as constants, for example, to set up the series  $1^3, 2^3, \dots, 100^3$  in  $A_1 \rightarrow A_{100}$  in translation we would have:-

```
:LIST
((O  $\rightarrow$  N)DO(100)
N:STAND  $\rightarrow$  @* @* @  $\rightarrow$  AN
1ON(N) REPEAT)
*
```

The above<sup>2</sup> would appear on the programme tape, and the listing of  $1^3 \rightarrow 100^3$  would occur as part of the translation process. However, it is also very convenient to be able to extract information from the machine once a programme has run and, perhaps, got lost. This can be done; For Example:

```
:LIST('
'A:/'
'B:/'
N:1)
*
```

Instead of using the START instruction e.g.

```
:START(1,ABC)
```

we could use

```
:LIST(:GOTO(1,ABC))
```

when the programme would be obeyed at (1,ABC) immediately.

In these "temporary" programmes the orders are translated into the machine in the normal way and when the closing bracket is encountered the orders are obeyed. After the final order is obeyed control passes back to the LIST input, and any further programme or local variables will overwrite the "temporary" programme hence no space is wasted.

A LIST is terminated by \* , control then passing back to the initial instruction input.

Various other characters occurring under LIST effects of a trivial nature; these and the previous facilities are summarised in the following table:- (Table 1)

Table 1: Effect of characters occurring under LIST

$A \rightarrow Z @$	with numerical suffix is address of variable or integer or machine orders.
$\rightarrow$	with numerical suffix is address of integers or machine orders in absolute machine location.
$. + - 0 \rightarrow 9$	introduce integer or variable. <u>Wrong</u> if address not set.
$<$	introduces machine code block. <u>Wrong</u> if address not set.
( )	Contains H-Code orders to be obeyed "there and then".
' '	Contains characters which are to be copied directly onto output tape.
)	Computer STOPS.
::	Computer WAITS. Carries on when button 1 on Address 2 on the keyboard is changed.
= CR LF SpSp	are ignored.
/ > ,	are <u>Wrong</u> .
*	closes LIST

#### 4.1. DATA

The instruction DATA occurs in the body of a procedure and is obeyed dynamically, It has exactly the same effect as LIST except that it reads off data tape in running. The final \* will pass control back to the running programme.

N.B. LIST can occur in the body of a procedure (except for H-Code orders in ()) but this is to be discouraged. It is a relic of an early edition of the translator.

<sup>2</sup>should N start at 1 or just loop 101 times



---

## 5. THE BODY OF A PROCEDURE OBEYING A PROCEDURE

### 5.1. THE BODY OF A PROCEDURE

Procedures are introduced by an instruction such as

:DEFINE (ABC)

where ABC is the name (which has been SET under SETR)

The procedure itself is terminated by the instruction

:CLOSE

In translation control then passes back to that part of the translator which deals with the organisational instructions whence more procedures or organisational instructions will be expected.

### 5.2. OBEYING PROCEDURE

The instruction, such as

:START (3, ABC)

stores the first order at which the programme is to be obeyed. When the computer is entered at 513 the computer will start obeying the programme at this point.

(3, ABC) refers to REFERENCE 3 (q.v.) of procedure ABC. Note that we can only start obeying a programme at a reference number. The first order of a procedure is reference 0 although it can not be labelled as such. i.e. (0, ABC)

:START (3000)

or

:START3000

will start at absolute machine location 3000.

:START (N3)

will start at the order indicated by the value of  $N_3$  when it is obeyed. ( $N_3$  is a SUFFIX (q.v.)).

Similarly

:START (M, ABC)

but this will be rarely used.

The START instruction can occur anywhere in the programme outside the procedure.

## 6. BINARY

This will cause a binary copy of the H-Code programme in the computer to be output.

To input the binary tape the translator should be in the machine and the normal entry to translate used.

If the 40 button on F1 is not depressed the binary tape is compared on input with the programme currently in the machine. This serves as a check that the programme has been correctly binarized.

As BINARY only outputs non-zero locations it is advisable to use the instruction Z to clear the whole of the programme store before translating a programme which is to be binarized.

When the binary tape is re-input then the machine is left in exactly the same state as when it was output. Hence new procedures and data can be input, corrections can be made etc.

As the translator is not output by BINARY any variables which overwrite parts of the translator with such an order as

:LS (N: 200)

can not be output by BINARY. Therefore if these variables are set by a LIST instruction then they will not be binarized, therefore the LIST will have to be input separately.

## 7. SUNDRY ORGANISATIONAL INSTRUCTIONS

### STORE

The machine will print out (preceeded by LF) the number of unused locations. (terminated by \*).

### Z

All the locations in the machine, except those occupied by the translator, will be cleared (i.e. put equal to zero).

### ::

The computer will wait - until the last button on the keyboard (1 on Address 2) is changed from its current position.

### ' , '

All characters between the inverted commas will be copied directly onto the output punch.

## 8. ERRORS IN PROCEDURE

If a coding error is detected by the translator in the body of a procedure i.e. between the `DEFINE ( )` and the final `CLOSE`, then that part of the procedure already translated will be, in effect, erased automatically and the corrected version of the procedure can then be translated without any wastage of space.

If an error in a procedure is discovered by the programmer after the procedure has been fully translated, he must take steps himself to erase the previous version of the procedure before attempting to translate the new one. This he does with an order such as

`:X(ABC)`

which will erase reference to the previous version of ABC. Then follows the new version of ABC beginning with, as normal

`:DEFINE(ABC)`

The X cancellation does not allow re-use of the space used by the original version of ABC therefore the "re-defining" of procedures is conditional on there being sufficient store left in the machine.

## 9. SUMMARY OF INITIAL (OR ORGANISATIONAL) INSTRUCTIONS

See: Table 2

Table 2: Summary of initial (organisational) instructions

Group A	SET	Sets up machine to receive new programme
Group B	SETV( , )	Sets variables in permanent and local alphabets
	SETS( , )	Sets integers in permanent and local alphabets
	SETR( , )	Sets procedures with given names and given references
Group C	DEFINE( )	Introduces the body of a procedure
	LIST	Lists of variables and blocks of H-Code
	LV( , )	variables and integers respectively:-
	LS( , )	A, sets variable in local alphabet
		$A \rightarrow B_{20}$ , A in local alphabet takes properties of $B_{20}$ in permanent
		A : 3000, A in local alphabet becomes absolute location 3000
	PL	Permanent alphabet to local alphabet
	LP	Local alphabet to permanent alphabet
	STORE	Prints store left
	START( )	Stores reference and procedure for starting at 513
Group D	BINARY	Binarizes the programme in store
	X( )	Cancels the given procedure
Group E	Z	Clears that part of store used by programme and data
	::	Wait
Group E	' '	Title between is copied

### Notes

The first word of Group B which occurs before a word of Group C has also the effect of Group A i.e. SET.

## 10. FLOATING POINT FORM OF NUMBERS

### 10.1. FLOATING POINT FORM

A common way of writing very large or very small numbers in scientific work is to associate a power of 10 with each of them; for example:

- 35,742,000,000,000,000 may be written:
- $3.5742 \times 10^{16}$
- or 0.000,000,000,000,123,4 may be written:
- $.1234 \times 10^{-12}$

That is to say the numerical part, or argument, or mantissa is written with the decimal point in a convenient position and the exponent of 10 is an integer adjusted to make the overall value of the number correct. This way of representing numbers is generally called floating-point form, as opposed to fixed-point form of numbers written without an exponent. Thus within a computer a number can be represented in this form by 2 quantities, the argument (a fraction) and the exponent (an integer) packed into one computer word — or zero.

When a number is printed out by methods (ii) and (iii) in PRINT STATEMENT it is printed out in what is called Standardised Floating-point form. In this if  $A10^b$  is the number then A and b are adjusted so that if  $A \neq 0$  then  $.1 \leq A < 1$  when b is positive or negative integer.

---

if  $A = 0$  then  $A = 0$  and  $b = -78$  ( $b$  is irrelevant in fact).

Users need not be conscious of the form of floating-point except when they are printing variables in floating point form – and to a lesser extent when preparing data in this form.

## 10.2. FORM OF NUMBERS

### (a) In Arithmetic Orders

In arithmetical statements a variable can be replaced by an unsigned constant (except, of course that we cannot write  $\rightarrow 23$ )

Integers must not have decimal points, and they must be less than  $2^{38}$  i.e. 274,877,906,944

Floating point numbers should, in general, have a decimal point, however if the mode of the part of the expression has been determined the decimal point may be omitted. The integral part of the number must be less than  $2^{38}$  and the 'integer value' of the fractional part (i.e. the fraction treated as an integer e.g. .0001234 would be 1234) also less than  $2^{38}$ . An exponent is not allowed.

A number in arithmetic is terminated by the first non-numeric character.

### (b) In data for READ, LIST, DATA

In data a negative number must be signed, a positive sign is optional.

Integers as in (a)

Floating point numbers as in (a), but decimal point is not necessary in integral numbers. A decimal exponent is allowed – the argument is followed by a / followed by the decimal exponent which is a signed or unsigned integer  $\leq 77$  in absolute magnitude (the maximum number allowed in the machine is approximately  $(\pm 1.16 \times 10^{77})$ )

e.g. -13/-2 or 2.63/7

A number in data must be terminated by CR, spsp or LF.

### (c) In machine code blocks

In machine code the number must be signed.

Integers as in (a)

Floating point numbers as in (a), but the decimal point must be used. An exponent is not allowed.

A number in machine code must be terminated by LF or, in the case of integers, by <

## 11. MACHINE CODE BLOCKS

Blocks of machine code orders occur in the body of a procedure amongst the ordinary mnemonic statements where they are translated and obeyed in sequence with the other statements.

They also occur under LIST outside (or inside) a procedure where they are stored in the address as specified by LIST.

Although the full T102 (Elliott machine code input. routine) facilities are not available enough are provided for the vast majority of uses of machine code. The main difference is that fixed point fractions are not allowed. We are restricted to integer and floating-point working.

A block of machine code is introduced by < and closed by )

In the block we are allowed the following types of words:–

### Normal orders

$p_1q_1N_1/p_2q_2N_2$

$/p_2q_2N_2$

$p_1q_1N_1/$

$/$

where / can be replaced by :.

$p$  and  $q$  are the integers  $0 - 7$  and  $N$  is of the forms

123                      i.e. (integer)

123,                      , is relative to first order of

-123,                      block

-123

A12 or @12

(3) or (3,ABC)      i.e. the jump suffix (use with  
40 – 43 instructions)

### Octal

8 followed by the 13 octal (0-7) elements of the word.

### Numbers

---

Introduced by  $\pm$ . The presence, and only the presence, of a decimal point will make the number into a floating point number - otherwise it will be an integer.

In floating point number / followed by decimal exponent is not allowed.

Fixed point fractions are not allowed. An order such as

A12

will represent the address of  $A_{12}$  as an integer.

- An integer followed by , will give the integer plus the address of 1st order of block

e.g. +13,  
1,

New address (Allowed only under LIST)

An integer followed by < will alter the address for subsequent, orders — but , will still be relative to the beginning of the block. e.g.

A12<

+13<

-1,<

An order must be closed by LF or )

Systems of Blocks of machine code are not the same as T102. Each block is 'put into a letter' and reference made from one block to another with an order such as

73A:40A1

which would use the block in A as a subroutine. An order such as

730,2:401,2

has no meaning as Block 2 is not defined.

This system will not effect the use of library routines as blocks of programme as they generally only refer to themselves and not to other blocks. The only stumbling block to full use of Elliott machine code library routines is probably the lack of fixed point fractions and these appear to be rarely used - if at all.

## 12. USE OF PROCEDURES

A programme is made up, in general, of a group of procedures each of which does its own job using the other procedures as subroutines or passing control to them in turn.

By direct entry

In this method control is passed from the current procedure to another by 'jumping into' it with an order such as

: IF (X) ZERO : GOTO (3, ABC)

Here control passes to beyond reference 3 of procedure :ABC. No link is set - i.e. no marker is kept of the position or the procedure from which control was passed.

As a Subroutine

In this method control is passed to the Subroutine but a link is set and when the statement

: EXIT

occurs in the subroutine control is passed back to the order after the statement which calls on the subroutine, which is typified by

: ABC (N3+1)

i.e. the name of the subroutine procedure together with a simple suffix referring to the reference number at which entry is to be made. Diagrammatically:—

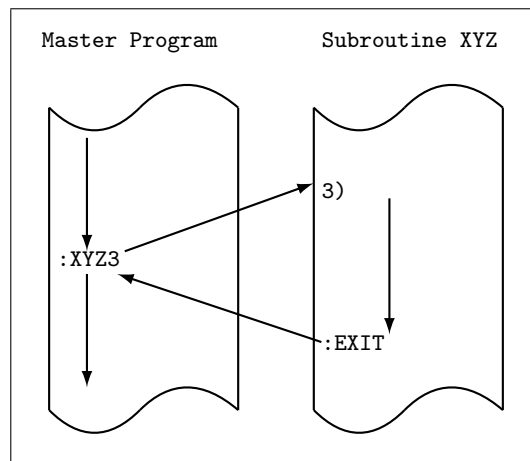


Figure 1: The use of a subroutine

As there is only one link associated with each procedure a procedure can not be entered as a subroutine from itself or from any sub-sub.-routine etc., of that procedure. Although there is no objection: to using a part of the same procedure as a subroutine if exit from the 'master part' of the procedure is not by obeying link (i.e. by EXIT). (i.e. no recursive use of procedures). Apart from the above restriction there is no limit to the depth of subroutine. Although in practice it is, of course, limited by the number of procedures which make-up the programme as a whole.

### 13. STRUCTURE OF A PROCEDURE

The basic structure of a procedure (ABC say) is

```
:DEFINE(ABC) , , , , , :CLOSE
```

Between the commas are statements — such as

Arithmetical statements

Conditional statements

Machine Code Blocks

Read Statements

Defined Functions

etc

etc

However, many statements themselves imply comma, so that we can string such statements together without actually using a comma. Similarly line-feed implies comma hence statements can be put on separate lines without commas.

The omitting of commas can make the print-out look tidier, but care should be taken to ensure that when a word implies a comma and the new statement begins with a letter then there should be some non-alphabet character between the two statements — e.g. comma itself or spsp. e.g. we can write

```
: IF (A=B) THEN 1.0 →C: ,
```

but we cannot write

```
: IF (A=B) THEN X →C: ,
```

as the X would be taken to be part of THEN (see WORDS). We can get over this in many ways. e.g.

```
: IF (A=B) THEN , X →C: ,
: IF (A=B) THEN (X →C) : ,
: IF (A=B) THEN X →C: ,
```

or

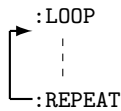
```
: IF (A=B) THEN
X →C: ,
```

Details of those statements which imply comma are given in the summary of facilities.

### 14. LOOP AND DO INSTRUCTIONS

The Instructions:—

(i)



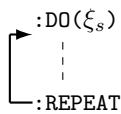
which must always occur as a pair facilitate the programming of repetitive processes. The programme between the LOOP and REPEAT is obeyed repetitively, the REPEAT instruction sending control back to the instruction immediately following the LOOP instruction. LOOP has no effect other than introducing a loop.

(ii)



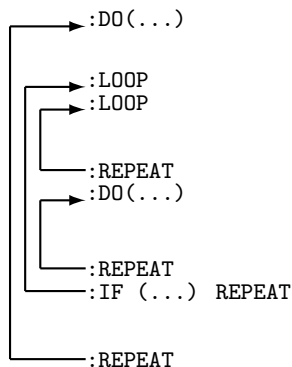
This is similar to (i) except that the REPEAT is conditional. The IF can introduce any kind or complexity of conditional statement.

(iii)



The programme between the DO and the REPEAT is obeyed  $\xi_s$  times — i.e. the value of  $\xi_s$ , when the programme is obeyed, The REPEAT associated with a DO must be unconditional.

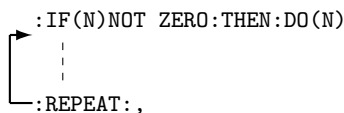
LOOP loops and DO loops can occur 'inside' other loops to a depth of 9. A REPEAT is paired with the last impaired LOOP or DO. e.g.



Other programme can occur anywhere between the LOOPS DOs and REPEATs.

**Note (i)** If  $\xi_s$ , in  $DO(\xi_s)$  is negative then there will be error signal.

If  $\xi_s$  is zero then it will be equivalent to  $DO(1)^3$ . However if we want literally to DO a loop zero times i.e. by-pass the loop we would write something like:



(ii) **Example**

Calculate

$$C = \sum_{I=0}^9 A_I B_{2I}$$

This can be done in various ways:

e.g. (a)

```

0.0 →C
(0 →I →I1)DO(10)
(AI+BI1)ON(C)
10N(I, I1, I1) REPEAT

```

<sup>3</sup>handwritten note: "ERROR" here

---

or (b)

```

0.0 →C
(0 →I)DO(10)
(AI+B'(I+I))+C →C
1ON(I)REPEAT

```

## 15. ON STATEMENT

The statement:—

$\xi$ :ON(P,Q,R,...)

where:

$\xi$  is fixed or floating point expression

$\xi$ , P, Q, R... are all same mode where P, Q, R... are any number and can have simple or complex suffixes.

Is such that the function ON adds the value of the accumulator onto each of the variables specified on the right hand side.

e.g.

$(N+M)ON(A3, B'(P+Q))$

will perform the operations

```

N+M+A3 →A3
N+M+B'(P+Q) →B'(P+Q)

```

and

$1:ON(L,M,N)$

is equivalent to

```

L+1 →L
M+1 →M
N+1 →N

```

## Notes

(i) With integers ON is always more efficient than the appropriate arithmetic. But with floating point variables its use, as far as efficiency is concerned, becomes debatable except when  $\xi$  is complicated or if any of the RHS have complex suffixes.

(ii) ON implies comma except that after final ) a letter-introduces a word. This is to allow the statement such as

$1:ON(N)REPEAT$

which is the most common use of ON.

(iii) After , or ( CR,LF, spsp are ignored.

## 16. SUFFICES

Integers or integer expressions can be used as suffices to floating- or fixed-point variables.

### 16.1. SIMPLE SUFFICES

The following examples cover all the allowable types of SIMPLE SUFFICES.

```

A13    or   A(13)
AN      or   A(N)
AN15    or   A(N15)
A(N+20)
A(N5-6)

```

Simple suffices can be written in this and only this form. e.g.

$A_{(n_1+20)}$

must be written as:

---

$A(N1+20)$

$A(20+N1)$  will not be accepted.

$A_{N15}$

must be written as:

$AN15$     or     $A(N15)$

$A(N(15))$  will not be accepted.

## 16.2. COMPLEX SUFFICES

These are written

$A'(\xi_s)$

$\xi_s$  itself can contain integers or variables which have simple suffices themselves, but not complex suffices

Examples:

$A'(P1+Q2)$   
 $A'(B:INT)$   
 $A'(NM3+N(M4+1)-5)$   
 $I'((N+M)*(N-M))$   
 $I'((N+M)SQ)$

## Notes

- (i) It is important that the value of a suffix is  $\geq 0$  and  $\leq 8191$  as values outside this range cause the arithmetic order to be destroyed.
- (ii) A suffix should be represented as a simple suffix where possible as complex suffices are less efficient in time and space.
- (iii) Complex suffices can be used in all types of arithmetic except division by an integer. For details see Arithmetic.
- (iv) As teleprinters have no multi-level printing facilities suffices are printed on the same level as the variable which they qualify but it should be remembered that  $A4$  means  $A_4$

## 17. JUMP SUFFICES, UNCONDITIONAL JUMPS

### 17.1. JUMP SUFFICES

The jump suffix (§) can be of the following forms

- (i) as an ordinary SIMPLE SUFFIX

- (ii)  $(3, ABC)$   
 $(N4+5, ABC)$

Jump suffices are used to qualify the `START` and `GOTO` statements.

Type (i) will refer to reference numbers within the current procedure, or if used outside a procedure, to the absolute machine location.

Type (ii) will refer to reference numbers in the specified procedure.

### 17.2. UNCONDITIONAL JUMPS

The statement

`:GOTO (§)`

will cause control to pass to the order immediately following the reference number specified by (§) (see JUMP SUFFIX above).

e.g.

`:GOTO3`  
`:GOTO(N, ABC)`  
`:GOTO(N)`

(note that: `GOTO N` would be wrong as `N` would be taken as part of the word `GOTO`).



## 18. CONDITIONAL STATEMENTS

There are two methods of writing conditions, one of which is more efficient than the other (in so far as it uses less space and is dynamically faster). The less efficient method is used because it is rather easier to use when a programme is being written quickly - say for a one-off job, or when mathematical clarity is needed.

### Type 1 (Efficient)

( $\xi$ )NEGATIVE	<0
( $\xi$ )ZERO	=0
( $\xi$ )POSITIVE	>0
( $\xi$ )NOT NEGATIVE	=0 >0
( $\xi$ )NOT ZERO	<0 >0
( $\xi$ )NOT POSITIVE	<0 =0

### Type 2 (Less efficient than type 1)

( $\xi < \xi'$ )	
( $\xi = \xi'$ )	
( $\xi > \xi'$ )	
( $\xi : < \xi'$ )	= or >
( $\xi := \xi'$ )	< or >
( $\xi : > \xi'$ )	< or =

where  $\xi$  and  $\xi'$  are the same modes

$\nearrow$  is punched : <,  $\neq$  :=,  $\nearrow$  : >

### 18.1. GENERAL CONDITIONAL STATEMENT

If we write ( $\emptyset$ ) for the general condition (any of above) the general conditional statement is:-

```

: IF ( $\emptyset$ ) {
  AND } ( $\emptyset'$ )
  OR }
  REPEAT
  GOTO ( $\$$ )
  THEN

```

This means that a conditional statement is always opened with an :IF. After the IF follows the condition on which the statement acts. After the condition follows the instruction which is obeyed if the condition is true. It can be one of five words.

(i) AND

(ii) OR

These allow us to have statements which depend on the truth of two or more conditions. After the AND or OR the statement behaves as if there had just been an IF. This is indicated by the arrow in the statement of the general conditional statement above.

Examples:-

```

: IF (X<A+B) AND (Y) ZERO : AND (Z) ZERO : GOTO 3
: IF (X<A : SIN) AND (Y) ZERO : OR (X) POSITIVE : REPEAT
: IF (X=Y:MOD(4)) OR (Z:MOD<Y:MOD) THEN ... : ,

```

It is important to attach the correct meaning to these multiple condition statements. If the statement consists of all ORs then the statement is true if any one of the conditions is true. Similarly if the statement consists of all ANDs then the statement is true only if all the conditions are true. But the statement (e.g.)

```
: IF ( $\emptyset$ ) OR ( $\emptyset'$ ) AND ( $\emptyset''$ ) OR ( $\emptyset'''$ ) ... GOTO ( $\$$ )
```

would be interpreted as meaning

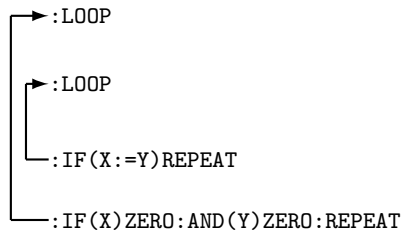
```
: IF ( $\emptyset$ ) OR [ ( $\emptyset'$ ) AND { ( $\emptyset''$ ) OR ( ( $\emptyset'''$ ) ... ) } ] GOTO ( $\$$ )
```

The conditions can not be grouped in any other way. If possible, the combination of ANDs and ORs should be avoided, or at least, kept very simple.

(iii) REPEAT

If the statement is true obey next the instruction immediately following the last unpaired LOOP instruction (see LOOPS OF INSTRUCTIONS).

Example:—



(iv) GOTO(\$)

If the statement is true obey next the instruction immediately following the reference number with current value (\$) (see JUMP SUFFIX).

Example:-

```
: IF (X=3) GOTO6
: IF (Y<B: SQRT) AND (Y) NOT POSITIVE: GOTO (N1+4, ABC)
```

(v) THEN

THEN is always associated with a ; (punched : ,) or an ELSE and a ; in the following manner:

- (a) : THEN , , , , : ,
- (b) : THEN , , , , : ELSE , , , , : ,

The commas (,) indicate that there can be statements between THEN, ELSE and ;

- (a) If the statement is true then those statements which occur between the THEN and the ; are obeyed. If the statement is not true then control passes to the statement immediately after the ;. e.g.

```
: IF (A<B*D) AND (C) ZERO: THEN , , , , :
```

- (b) If the statement is true then those statements which occur between the THEN and the ELSE are obeyed. If the statement is not true then those statements between the ELSE and the ; are obeyed. e.g.

```
: IF ((A+B)*(A-B)-8) NEGA: THEN , , : ELSE , , , , :
```

Any of the statements between THEN, ELSE and ; can themselves be THEN statements. This can occur to a depth of 9.

An ELSE is paired with the last impaired THEN. A ; is paired with the last unpaired THEN or ELSE.

Example:-

```
: IF (A*A-(B*C) →P) NEGATIVE: THEN
-P: SQRT →Q, -1 →U
: ELSE
: IF (P) ZERO: THEN , 0 →U: ELSE, P: SQRT →P, 1 →U, : ,
```

LOOP or DO ..... REPEAT loops are quite distinct from THEN statements and their pairings are independent so that we could have the following:

```
(0 →N) LOOP: IF (A<BN) THEN , 1 ON (N) REPEAT: ,
```

where the pairings of LOOP ... REPEAT and THEN ... ; cross each other. The above "compound" statement is in fact a most useful table search.

Note

- (i) Before words NEGATIVE etc. and AND etc. CR, LF, Sp<sup>2</sup> are ignored.

- (ii) In obeying a "compound" conditional statement as soon as the truth or falsity of a statement is determined then the rest of the conditions are "skipped" e.g. in

```
: IF (A<B) OR (C<D) OR (X>Y : MOD) GOTO (6)
```

If (A<B) is true then the whole statement is true and the other conditions are not examined but e.g. in

```
: IF (A<B) AND (C<D) . . . . .
```

If (A<B) is true then because it is followed by AND the other conditions must be examined. However if (A<B) is false then the statement is not true and the other conditions are not examined.

## 19. BRANCH

BRANCH is a function of the accumulator and a jump instruction which has up to 3 jump suffixes. It transfers control according to the sign of the accumulator. In general it is written:-

```
ξ : BRANCH(R1, R2, R3)
```

where R = (\$) the general jump suffix (qv)

It has the effect:—

If value of	$\xi < 0$	then control passes to	$R_1$
..	$\xi = 0$	..	$R_2$
..	$\xi > 0$	..	$R_3$

The general case can be reduced to

```
ξ : BRANCH(R1, R2)
or ξ : BRANCH(R1)
```

In these cases if the first one or two conditions are not satisfied then no action is taken.

Examples:-

```
(N-M) BRANCH(3, (N4+3), (N+1, ROUTINE))
```

## Notes

- (1) If R consists of more than one "item" the brackets must be included. e.g. in the above example we must have  $(N_4 + 3)$ .

In particular, in the reduced case we must write (e.g.):—

```
A : FRAC-W : BRANCH((N5-6))
```

Note the two sets of brackets.

## 20. READ STATEMENTS

On obeying an instruction such as:

```
: READ(A, B, N, C)
```

the first number is read off tape and sent to A, the second to B, etc.

A, B, N, C can be variables or integers in any order and there can any number of them. They can have simple or complex suffixes

For form of punching of numbers to be input by the READ see FORM OF NUMBERS IN DATA.

For effect of characters in searching for numbers see table 3

## Notes

- (i) In writing the READ instruction after the variable which is to be read we can carry on with normal arithmetic, with the restriction that the variable to be read follows immediately after the ( or , e.g. we can write: (see (ii)).

```
: READ(A, B3+C →X, Y'(P+Q))
```

After the first comma; the number will be read, sent to B3, C will be added and result sent to X. We can not write:

```
: READ(A, (B+C)*X →Y)
```

---

because the B does not immediately follow the comma.

- (ii) after , or ( CR LF SpSp are ignored. This is so that a READ instruction which is very long can be broken at the end of a line of teleprinter page.

Table 3: Effect of characters met in obeying READ instruction

letter shift, figure shift blank tape CR. Sp Sp. Sp. LF =	are ignored
. + - digits	introduces a number
'	introduces a title (terminated by ') which is copied onto the output tape directly.
::	Computer waits. Continues after the last button of keyboard word is changed from its current position.
)	Computer stops.
All other characters	Wrong. Computer gives error signal.

## 21. PRINT STATEMENTS

if m and n are integer constants there exist the following print statements:-

- (i)  $\xi_v : m . n$  will print out the value of  $\xi_v$  with m digits before the decimal point and n digits after.
- (ii)  $\xi_v : m / \dots$  in floating point form with m significant figures.
- (iii)  $\xi_v : /$  equivalent to :8/
- (iv)  $\xi_s : m \dots$  with m digits

All numbers are preceded by a negate sign if negative but by a space if not negative. No terminating symbol is punched after the number. If the numbers are to be re-input then it is necessary to punch the terminating symbol by use of the title instruction (q.v.). All numbers are preceded by a figure shift.

Non-significant leading zeroes are replaced by spaces.

In

- (i) If  $m > 12$  or  $n > 11$  then method (iii) is used.
- (ii) If  $m > 8$  .. .. (iii) .. ..
- (iv) then print with  $m = 12$ .

In (i) (ii) (iv) if m is not large enough then an attempt is made with  $m = m + 1$  until new m is big enough or until the above restrictions apply. The number is always printed somehow.

Fractions are rounded off by the addition of  $-5 \times 10^{-n}$  to the number in (i) and  $.5 \times 10^{-m}$  to the mantissa in (ii) (iii).

In

- (i)  $m + n + 2$  characters are printed (+ shift)
- (ii)  $m + 6$  ..
- (iii) 14 ..
- (iv)  $m + 2$  ..

Note

- (i) The print instruction does not preserve the accumulator and therefore can not occur in the middle of an expression.
- (ii) m and n must be integer constants e.g. :6.3 We can not write :P.Q.

## 22. REFERENCE NUMBERS

It is desirable to be able to start obeying a procedure or to break sequence in a procedure at a point other than the first order. For this reason we label statements in the following manner, e.g.

12) X+Y →Z

That is, a label is an integer constant written with a bracket before a statement. To avoid confusion there is a restriction that the label can occur only at the beginning of a new line on the teleprinter page. A label is also known as a reference number.

Control passes to the above instruction, say, (which occurs e.g. in procedure :ABC) by orders of the following types:

- (i) :IF(A) ZERO:GOTO(12)
- (ii) :GOTO(12,ABC)
- (iii) :ABC(12)
- (iv) :START(12,ABC)

- (i) is a jump from a statement within the same procedure (see CONDITIONAL STATEMENT)
- (ii) is a jump from a statement within another procedure (see UNCONDITIONAL JUMPS)
- (iii) calls on ABC as a subroutine entering at 12 (see SUBROUTINE)
- (iv) the whole programme is started at this point (see START)

A reference number can also be used in conjunction with TRACE to give indication of the path of the programme.

---

## Notes

- (i) All instructions in the same procedure to which reference is to be made must be given different reference numbers.
- (ii) Any reference numbers may occur, they need not occur in sequence, and the numerical sequence of references need not be complete.
- (iii) The first order of a procedure is automatically taken to be reference 0 (i.e. zero) but we must not actually label it 0).
- (iv) The highest reference number must be SET when the procedure is SET at the beginning of the programme (see SETTING OF PROCEDURES).

## 23. BRACKETS OR DEPTH OF EXPRESSIONS

A pair of brackets ( ) in an expression or statement gives the expression a depth of 1. If there is a pair of brackets within this pair then the expression is of depth 2 ...etc. There is a maximum allowable depth of 9.

All brackets except those

- (i) around simple suffices
- (ii) in ON( ) instructions, and
- (iii) in BRANCH( ) instructions, increase the depth of expression by 1

Note that

```
: IF (A / (C * D + (E * F))) ZERO : GOT01
```

is of depth 3 but that

```
A / (B * (N + M) + C * (N - M))
```

is of depth 2.

That is, as the expression is read from the left a ( will increase the depth by 1 but the corresponding ) will decrease it accordingly.

## Notes

- (1) In arithmetic expressions the depth should be kept to a minimum, disregarding for this purpose depths due to brackets used merely for decoration.

e.g.  $y = d(a + bc)$  is more efficiently written

```
(B * C + A) * D → Y
```

than

```
D * (A + (B * C)) → Y
```

It can be seen in the first case all the arithmetic is done directly in the accumulator but in the second the accumulator has to be stored away twice in order to perform the arithmetic.

It should be noted that what is defined as depth in RDR/100/3.4 is not strictly what is defined as depth here.

## 24. CONSTANT SEARCHING

In translation, copies of constants which occur in arithmetic (but not DATA or LIST) are kept in a list. If the same constant appears many times it is obviously wasteful in space to have numerous copies of that constant. For this reason a facility is provided for "constant searching". In this a search is made of constants already in the list before a new constant is added. Against this there is the fact that searching can slow down translation considerably. For this reason it is optional

If the 40 button in F1 on the number generator is depressed constant searching will not take place and vice versa.

As this button will normally be depressed constant searching will not take place unless required.

Only the positive value of a constant is put in the list. Integers 1-27 inclusive are treated separately and appear in a separate list in which constant searching always takes place.

---

## 25. CHECK

The function of the accumulator known as CHECK and written

$\xi : *$

is available in both fixed and floating point modes. It provides optional print-out of intermediate results. It is often very useful if CHECKS are included at suitable points of programmes, to assist in detecting programming errors, etc.

If B = 1 is set on the keyboard during the running of a programme, CHECK will cause the contents of the accumulator to be printed out on a new line preceded by \* and with the following print specifications:

variable as

$:/$

integer as

$:1$

CHECKS are only translated if B = 1 is set on the keyboard during translation. If B = 0,  $:*$  is ignored. Therefore we have two opportunities of ignoring  $:*$ . In translation and in running. Obviously if it is ignored in translation it can not be used at all in running. As a programme is run or translated each  $:*$  is treated on its own merits, i.e. some may be ignored, others assembled or used, depending on the state of B at that moment.

$:*$  does not affect the accumulator therefore it can be used in the middle of arithmetic expressions. e.g.

$((A+B) : * : \text{SQRT}) * C \rightarrow X$

## 26. TITLE AND OPTIONAL TITLE

### 26.1. TITLE

The statement which consists of a string of characters enclosed by two inverted commas, is known as a TITLE statement. When obeyed the characters between the inverted commas (dashes) will be printed out on the output tape.

e.g.

'LBS/FT3'

All characters are copied except blank tape which is ignored. All figure shifts or letter shifts whether redundant or not are copied. Every TITLE output is preceded by figure shift.

It is important to note that a TITLE statement is the only method (other than machine orders) by which any characters can be output. In particular, if we want a number to be printed on a new line we must include a TITLE which consists of CRLF, i.e.

;

### Notes

- (1) If say n blanks are to be punched, this must be done with the order

$:DO(N) < 740 : : REPEAT$

### 26.2. OPTIONAL TITLE

This statement is similar to above except that they are punched (e.g.)

$: 'CONVERGE'$

i.e. with a colon before the normal title instruction. These titles will only be output if the F2 40 button on the keyboard is depressed in running. They are always input on translating.

## 27. TRACE

The TRACE is used to indicate which course a programme is running. This is useful if, for example, the programme "gets lost". This it does by printing out, on a new line, the value of a reference number as control "passes through" it. As reference numbers are meaningless unless qualified by the procedure which they are in the form of printing is (e.g.)

13, ABC

i.e. reference 13 of procedure ABC, As a procedure is identified by the first 4 (or less) letters of a word no more than 4 letters are printed out. The reference number itself has the print specification

:4

Similar to CHECK, the TRACE facility is accepted in translation by depressing the F2 01 button on the keyboard, and the TRACE is printed out on running by depressing the same button. As the TRACE facility works in conjunction with reference numbers no extra characters have to be punched on the programme tape.

### Note

Since the first order of a procedure is understood to be reference 0 automatically, then on entering a - procedure at the 1st order the TRACE would be e.g.

0, ABC

## 28. ARITHMETIC STATEMENT

Arithmetic is left associative, i.e. if we write (in H-Code):

A+B\*C →Y

this is interpreted as meaning  $y = (a + b)c$  and not  $y = a + (bc)$ . This is because the expression is - interpreted as each operator is encountered - reading from left to right. No reference being made to the next operator the normal convention of multiplication and division being "stronger" than addition and subtraction does not apply.

The standard functions of analysis (sin, sqrt, etc.), are also left associative in the sense that they are a function of the quantity on their left rather than the right, e.g. the equation  $y = \sin(a + b)$  would be written:-

A+B: SIN →Y

The general arithmetic statement is:-

(-) A1:FCT&A2:FCT&A3:FCT...&AN:FCT

where A is any variable or constant or is itself an arithmetic statement enclosed in brackets

FCT is any standard algebraic function.

£ is any one of the operators + - \* / → but →A is not allowed if A is an expression in brackets or a constant. (see DEPTH OF EXPRESSIONS).

Example:-

$$i = ae^{-\frac{Rt}{2L}} \cos(\omega t + f)$$

can be written:

(W\*T+F) COS\*(-R\*T\*.5/L:EXP)\*A →I

## 29. ARITHMETIC OPERATIONS

The arithmetic operations

+ - \* /

are available in both fixed floating point

Integer multiplication will give an incorrect result if the result is greater than  $2^{37}$  in absolute value. The error will not be detected if overflow (see below) does not occur.

Integer division will give the correct answer if the division has an exact result, if not, it will give the largest integer not greater than the correct answer.

e.g.  $6/2 = 3$   
 $-6/2 = -3$   
 $2/3 = 0$   
 $15/4 = 3$   
 $15/(-4) = -4$



---

OVERFLOW is said to take place if the result of an arithmetic operation is outside the range of the machine (see Form of Numbers)

Floating-point overflow causes the computer to stop automatically.

Fixed-point overflow causes the computer to carry on using the wrong result.

Notes

- (1) Because of the nature of the translator direct division by an integer which has a complex suffix (q.v.) is not acceptable.

e.g. the term, where N & M are integers

$$N/M \text{ ' } (P+Q)$$

will not be accepted. If necessary, however, we can write

$$N/(M \text{ ' } (P+Q))$$

### 30. ALGEBRAIC FUNCTIONS

These functions occur anywhere inside arithmetic statements. The operand is in the accumulator and the result is left in the accumulator.

a) Functions valid for both variables and integers

- (i)  $\xi : \text{SQ}$  Forms the square of  $\xi$
- (ii)  $\xi : \text{NEG}$  Negates  $\xi$
- (iii)  $\xi : \text{MOD}$  Forms  $|\xi|$  i.e. the numerical value of  $\xi$   
e.g.  $|-3| = 3, |0| = 0, |3| = 3$ .

b) Functions valid for floating-point variables only

- (i)  $\xi_v : \text{SQRT}$  Forms  $\sqrt{\xi}$  where  $\xi \geq 0$
- (ii)  $\xi_v : \text{SIN}$  Form the trigonometrical
- (iii)  $\xi_v : \text{COS}$  functions of  $\xi$  where  $\xi$  is in
- (iv)  $\xi_v : \text{TAN}$  radians and  $|\frac{\xi}{\pi}| < 2^{28}$
- (v)  $\xi_v : \text{ARCTAN}$  Forms the angle, in radians, whose tangent is  $\xi$  such that it lies between  $\pm \frac{\pi}{2}$
- (vi)  $\xi_v : \text{EXP}$  Forms the exponential of  $\xi$ ,  $e^\xi$ , where  $\xi \leq 254 \log_e 2$
- (vii)  $\xi_v : \text{LOG}$  Forms the natural logarithm,  $\log_e \xi$ , where  $\xi > 0$
- (viii)  $\xi_v : \text{FRAC}$  Forms the fractional part of such that  $\text{frac}(\xi) + \text{int}(\xi) = \xi$  therefore e.g.  $\text{frac}(3.4) = .4$ ,  $\text{frac}(-5.4) = .6$ ,
- (ix)  $\xi_v : \text{INT}$  Forms the integral part of  $\xi$  and leaves it in integer form. The integral part is the largest integer not greater than  $\xi$ . e.g.  $\text{int}(3) = 3$ ,  $\text{int}(3.4) = 3$ ,  $\text{int}(-3.4) = -4$ .

INT is the only way of changing the mode of an expression from floating point to fixing point.

c) Functions valid for fixed-point variables only

- (i)  $\xi_s : \text{STAND}$  Converts the integer value  $\xi$  to the floating-point variable of the same value, STAND is the only way of changing the mode of an expression from fixed point to floating-point.  
or  $\xi_s : V$
- (ii)  $\xi_s : \text{MOD}(\xi'_s)$  Forms  $\xi(\text{modulo } \xi')$  i.e. gives the remainder when  $\xi$  is divided by  $\xi'$ .  $\xi' > 0$  and result  $\geq 0$ .  
e.g.  $7(\text{modulo } 3) = 1$ ,  $-7(\text{modulo } 3) = 2$ ,  
 $8(\text{modulo } 2) = 0$ . In fact, it performs the fixed-point operation  $\xi - (\frac{\xi}{\xi'} * \xi')$ . As MOD( ) is a subtraction rather than a division process the function should be used with care because if  $\frac{\xi}{\xi'}$  is very large then the time of operation will become very long. This function is distinguished from MOD in a) (iii) above, with which it has no connection, by the following brackets.

Notes and Examples

- (i) the argument of SIN, COS, and TAN can be any angle (subject to  $|\frac{\xi}{\pi}| < 2^{28}$ ). The SIN for example, is taken as the angle modulo  $2\pi$ .

- (ii) to find y say, the integral part of A, where y is floating point we have to write

$A : \text{INT} : \text{STAND} \rightarrow Y$

(see (viii) below)

- (iii) In order to raise a quantity to a power which is not a whole number, e.g. to form

$$x = a^b \text{ we would write}$$

$A : \text{LOG} * B : \text{EXP} \rightarrow X$

- (iv) In order to save time in computing, it is often possible to avoid calculating a function more times than necessary, for example, to programme the equation:-

$$y = \frac{e^x + e^{-x}}{2}$$

it is very easy to write:-

$((X : \text{EXP}) + (-X : \text{EXP})) / 2 \rightarrow Y$

but more economical to write:

$((X : \text{EXP} \rightarrow Y) * (1.0 / Y)) / 2 \rightarrow Y$

Note the use of y as an intermediate store for  $e^x$ , which avoids calculating an exponential more often than necessary.

- (v) The equation  $y = e^{-(x-a)^2}$  can be written either

$((-(X-A) \text{SQ})) \text{EXP} \rightarrow Y$   
or  $(X-A) \text{SQ} : \text{NEG} : \text{EXP} \rightarrow Y$

It is generally better to use NEG when negating an expression  $-(\dots)$  involves increasing the depth of the arithmetic which is to be avoided if possible.

(vi) it is useful to remember that  $\text{int}(A) + \text{frac}(A) = A$ . In fact finding  $\text{frac}(A)$  the computer finds  $A - \text{int}(A)$ . Therefore if both  $\text{frac}$  and  $\text{int}$  are needed it is better to calculate only  $\text{int}$ .

(vii) Functions to calculate arcsin and arccos are not provided but:-

$$\arcsin x = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

$$\arccos x = \arctan\left(\frac{\sqrt{1-x^2}}{x}\right)$$

e.g. for the arcsin of A we could write:-

$$A / (1.0 - (A * A) : \text{SQRT}) : \text{ARCTAN}$$

(viii) If we want to form  $y = n + x$  when  $x$  and  $y$  are in floating point form and  $n$  is an integer we would write:-

$$N : \text{STAND} + X \rightarrow Y$$

(ix)  $\text{MOD}()$  can be used to depth. e.g. we could write:-

$$N : \text{MOD}(M : \text{MOD}(L) + 2)$$

### 31. $\alpha$ - $\theta$ STORES

$\alpha$  is a floating point variable which, as well as possessing all the properties of other floating point variables, has the special property that if we write:-

$$\{\xi_v^{(i)}, \xi_v^{(ii)}, \xi_v^{(iii)}, \dots\}$$

then this is equivalent to writing

$$\begin{aligned} \xi_v &\rightarrow \alpha \\ \xi_v^{(i)} &\rightarrow \alpha_1 \\ \xi_v^{(ii)} &\rightarrow \alpha_2 \\ \xi_v^{(iii)} &\rightarrow \alpha_3 \\ &\vdots \\ \xi_v^{(n)} &\rightarrow \alpha_n \end{aligned}$$

or in general

for  $n = 0, 1, 2, \dots$

Similarly  $\theta$  is a fixed point variable such that

$$[\xi_s, \xi_s^{(i)}, \xi_s^{(ii)}, \xi_s^{(iii)}, \dots]$$

is equivalent to  $\xi_s^{(n)} \rightarrow \theta_n$ , for  $n = 0, 1, 2, \dots$

In general we can write (e.g.)

$$\{\xi_v, \xi_v^{(i)}; \xi_s; \xi_v^{(ii)}, \xi_v^{(iii)}; \xi_s^{(i)}, \dots\}$$

or

$$[\xi_s; \xi_v, \xi_v^{(i)}; \xi_s^{(i)}, \xi_s^{(ii)}, \dots]$$

which are equivalent to

$$\xi_v^{(n)} \rightarrow \alpha_n \text{ for } n = 0, 1, 2, \dots$$

$$\xi_s^{(m)} \rightarrow \theta_m \text{ for } m = 0, 1, 2, \dots$$

That is, the type of bracket determines the mode of the first expression, thereafter the mode is changed by ; (Although this indication of mode by type of bracket is not strictly necessary, it is required as a safeguard against careless programming, in particular against writing an integer in floating point without the decimal point).

The reverse procedure is (e.g.)

$$\rightarrow (A, A^{(i)}, N, A^{(ii)}, N^{(i)}, \dots)$$

where A are floating and N are fixed-point variables with simple or complex suffices is equivalent to:-

$$\alpha_n \rightarrow A^n \text{ for } n = 0, 1, 2, \dots$$

$$\theta_m \rightarrow N^m \text{ for } m = 0, 1, 2, \dots$$

Variables and integers can be mixed in any order and semi-colons between them are not necessary although they can be used if desired.

### Notes

(i) In the  $\rightarrow ()$  instruction as in the READ instruction (Note (i)) after the variable we can carry on with normal arithmetic.

- (ii)  $\alpha, \theta$  are set automatically by SETV, SETS to be  $\alpha_9, \theta_9$ . But if more used these must be set.
- (iii) These  $\alpha$ - $\theta$  statements ("nests") imply comma, except that after the final ) a letter introduces a word. this is so we can write (e.g.)

{A, B+C; N}ABC

where ABC is a word statement (usually a defined function). This is the most common use of a nesting store.

{ } is punched \*( )  
 [ ] .. :( )  
 ; .. ; ,

$\alpha$  if the teleprinter character @  
 $\theta$  is letter O.

- (iv) Example If there was a defined procedure QUAD which gave the two roots of a quadratic given the three coefficients the order to do this might look:—

\*(1.6, 3.6, -4.53)QUAD →(X1, X2)

### 32. BAR A $\bar{A}$

$\bar{A}$ , called BAR A, punched =A, is an integer which has the value

Absolute Address (A) -1

where A is any variable or integer.

If B, for example, has been set as

:LV(B:1) or :LS(B:1)

then if  $\bar{A}$  has been sent to N then

BN will be equivalent to A

This means that subroutines can be written in terms of any variables irrespective of what variables a master procedure will use. Part of the input data to a subroutine would be say,  $\bar{A}$ , telling the subroutine that such and such a parameter was A.

Arithmetic can be performed with  $\bar{A}$  etc. but we cannot write:—

=A--B

it must be

=A-(=B)

$\bar{A}$  cannot have a suffix, but if we wanted the address of  $A_3$  we would write

=A+3

### 33. SUNDRY INSTRUCTIONS IN BODY OF PROCEDURE

#### STOP

On obeying this order dynamically, the machine will STOP. There is no way of carrying on from that point.

#### WAIT

On obeying this order dynamically the machine will WAIT, until the Address 2: 1 button is changed (see KEYBOARD). The machine will then obey the next order etc.

#### CLOSE

Signifies that the procedure is closed.

#### EXIT

On obeying this order dynamically control will pass back to the master procedure which is using the current procedure as a subroutine (see USE OF PROCEDURES)

#### Line-feed

In translation line-feed has the same significance as comma which closes a statement. After a line-feed all subsequent lf, cr,  $sp^2$  are ignored until the next significant character.

#### Comma ,

In translation, closes a statement (see STRUCTURE OF PROCEDURE).

#### DATA

Accepts a data tape in the form of LIST.

#### : CR LF

Ignored in position of + in arithmetic statement

::

In translation the computer will wait - until the address 2:1 button is changed (see KEYBOARD). The machine will then carry on translating.

#### 34. TELEPRINTER CHARACTER: LAYOUT OF PROGRAMME ON TELEPRINTER SHEETS

Generally speaking except for misuse of carriage return without line-feed, if a programme 'looks right' on the teleprinter print-out then the tape which produced it will be correct also-and vice-versa. The following notes on the characters are true for all parts of the programme-procedures, data etc. expect<sup>4</sup> those character appearing between ' ' in title sequences.

blank-tape is always ignored.

Figure-shift

Letter-shift have no significance other than as shift characters. The use of more shift characters than necessary is allowed. Hence letter shift (ls) can be used as an erase character, by backspacing and over-punching the incorrect character with ls - then figure-shift (fs) if necessary.

? is never used

single space (Sp) is ignored everywhere but ...

double space (Spsp or  $Sp^2$ ) has significance as a terminating character to numbers, words etc. It has the same properties as carriage return (cr)

@ is understood to be  $\alpha$

letter O when used as an integer variable is taken to be theta ( $\theta$ ).

Title characters - i.e. those between ' ' are treated independently. Notes:—

blank-tape In a straight copy from input to output is copied but when a title is stored and output in running bl is ignored.

Shifts Are copied exactly as they appear on the input tape.

' Cannot be used in a title sequence.

Elliott Characters	£	correspond to	→
	\$		<
	%		>

#### 35. WORDS

A word can consist of 1, 2, 3, or 4 letters. If a word has 4 letters then extra letters can be added to the end if required - e.g. The word for repeat is REPE but we write this as REPEAT to make the printout more readable. These extra letters have no significance whatsoever. The translator in reading a word will take note of up to the first 4 letters and keep on reading, but ignoring, letter until the first non-alphabet character, which is taken to belong to the next operation.

To distinguish words from variables, all words are preceded by colon e.g.

: WORD

but where there is no ambiguity i.e. in a position where a letter can only introduce a word, this colon can be omitted e.g.

:GOTO(1,ABC)      instead    of    :GOTO(1, :ABC)

(A+B)SIN      ..      ..    (A+B):SIN

but we must have      A:SIN

:DEFINE(ABC)      ..      ..    :DEFINE(:ABC)

or any initial word

SET      ..      ..    :SET

But if in doubt always put in the colon

Throughout this manual the colon has been omitted where there is no ambiguity both in the English and in the examples of the coding.

#### 36. SUMMARY OF FACILITIES WITHIN PROCEDURE

See Table 4

---

<sup>4</sup>maybe should be "except"

### 37. USE OF COMPUTER KEYBOARD

See Table 5

Table 4: Summary of facilities within procedure

DEFINE( ).....CLOSE	Defines body of procedure	
SQ NEG MOD ON(...)*	Algebraic function of variable or integer	
SQRT SIN COS TAN EXP ARCTAN LOG FRAC INT	Algebraic function of variable only.	
STAND MOD (..)	Algebraic function of integer only.	
BRANCH( , , )	Jump on sign of LHS	★
:n.m :m/ :m :/	Prints	★
:*	Check	
Any defined function ABC	Defined function	★
LOOP....REPEAT DO( )....REPEAT	LOOPS	★
GOTO		★
GOTO IF .... REPEAT AND OR	Conditional statement	★
GOTO EXIT	Unconditional jumps	★
IF .... THEN .... ELSE ... ;	Then Clause	★
READ DATA	Input of numbers	★
STOP WATT	Running stops	★
[ ] { } →( )	$\alpha, \theta$ stores	★
3)	Reference number	★
' ' : ' '	Titles	★
< )	Machine-code block	★
LF ,	Close statement	

★ Those instructions imply comma.

#### Note

Although a defined function implies a comma, if it is immediately followed by a title statement without a comma to split them then this will be an error as the translator takes the ' as the introduction to a complex suffix, even though this is not allowed.

Table 5: Use of Computer Keyboard

FUNCTION 1

O

4<sub>1</sub>

O

2<sub>1</sub>

O

1<sub>1</sub>

O

4<sub>2</sub>

O

2<sub>2</sub>

O

1<sub>2</sub>

Buttons and number  
in square are used

ADDRESS 1

O

4096

O

2048

O

1024

O

512

O

256

O

128

O

64

O

32

O

16

O

8

O

4

O

2

O

1

O

B

FUNCTION 2

O

4<sub>1</sub>

O

2<sub>1</sub>

O

1<sub>1</sub>

O

4<sub>2</sub>

O

2<sub>2</sub>

O

1<sub>2</sub>

ADDRESS 2

O

4096

O

2048

O

1024

O

512

O

256

O

128

O

64

O

32

O

16

O

8

O

4

O

2

O

1

<u>OPERATION</u>	BUTTON	DOWN	UP
<u>INPUTTING BINARY TAPE</u>	Function 1:4 <sub>1</sub>	Input programme	Compare programme
<u>INPUTTING NORMALLY</u>	Function 1:4 <sub>1</sub> Address 1:512 Address 1:512 and :1 B Function 2:1 <sub>2</sub>	No constant search Entry to translate Entry to obey START i.e. 513 Translate check Translate Trace	Constant search    Ignore check Ignore trace
<u>RUNNING</u>	B Function 2:4 <sub>1</sub> Function 2:1 <sub>2</sub>	Print Check Print optional Title Print Trace	Ignore check Ignore Option Title Ignore Trace
<u>GENERAL</u>	Address 2:1	Stop waiting if up	Stop waiting if down

### 38. ERROR INDICATION

In translation and in running errors will be indicated by the continuous output of a letter. Although this letter will give some indication of the type of error encountered it should not be looked upon as a rigorous classification. A particular indication can be very misleading due to the very nature of an error. The following indications will be given.

#### (i) In Translation

##### Continuous letter—shift-letter

<u>Indication</u>	<u>Classification</u>
A	Incompatibility of variables and integers. Variables not set.
B	Incorrect characters at beginning of arithmetic expression
C	Incorrect character elsewhere in arithmetic expression
D	Error in data or number in body of programme
E	WORD wrongly used or not set
F	
G	
H	Error in DATA read
I	Reference too big or set twice. Depth of loops, arith. too deep. Programme too big.
J	Too many digits in integers, various odd mistakes
K	Excess of left or right-hand brackets in arith. depth of loop at CLOSE not zero.

#### (ii) In Running

##### continuous letter-shift-letter

- A Jumping to reference which has not been set or calling on procedure which has not been defined.

##### Continuous digit (or letter)

- 1 or A A in sin A, cos A or tan A too big  
(see appropriate section)  
2 or B  $A \leq 0$  in log A  
3 or S  $A > 254 \log_e 2$  in exp A  
5 or U  $A < 0$  in SQRT A

Notes (i) In translation if the computer 'gets lost' this is most probably because on a :: wait after a SET instruction the translator has been entered at the beginning instead of the wait being 'cancelled'.

(ii) In running if the programme does strange unaccountable things, this is most probably because the variable SET's have not been high enough and the programme is being overwritten by data.

#### Hand written notes:

##### Continuous letter-shift blank

DO(<1)

##### Modification

:I → Contents of accumulator



---