503
COMPUTER
MANUAL

VOL. 2
A

503 COMPUTER MANUAL

INTRODUCTION

This volume is intended to provide the programmer with all the information necessary to write programs for the 503. For a full account of the electronics, registers and similar facilities, the interested reader is referred to Volume 4 of the Manual. Certain operating instructions are included in this Volume, but Volume 3 of the Manual is intended for the operator of the computer.

The programmer or systems analyst using the 503 has available to him, in addition to the hardware of the machine itself, a considerable range of carefully planned programming systems, including internationally accepted problem–oriented programming languages, and at a more machine–oriented level, such items as macro–instructions for using peripheral equipment with maximum efficiency.

Such programming systems are made possible largely by the greatly increased basic speed of the machine relative to earlier electronic computers. They represent great advances in the ease of programming and the range of problems susceptible to data processing.

It is for the user to decide of which, if any, of these systems he wishes to use. The basic instructions of the machine are also described in full detail.

503 TECHNICAL MANUAL

VOLUME 2: PROGRAMMING INFORMATION

CONTENTS LIST

PART 1: PROGRAMMING SYSTEMS

SECTION 1: BASIC MACHINE PROGRAMMING

CONTENTS LIST

CHAPTER I

THE INSTRUCTION CODE

This chapter describes the 64 functions of the 503 Instruction Code. A summary of the instructions is in 1.2.1.

The examples in this chapter are written in terms of absolute machine locations to give a clearer idea of the use of the various functions. A program referring to absolute locations should not normally be used on the 503: S.A.C. (see 2.1.2) should be used instead. The Symbolic Assembly Program translates identifiers into absolute locations when a program is input under its control, using storage space available at the time. When programs are being written in S.A.C. the functions may be replaced by their mnemonic names (i.e. AR may be used instead of 00). In this chapter, the mnemonic name is given in brackets after each of the functions.

THE REGISTERS

There are seven registers of interest to the programmer:—

An Accumulator of 39 bits.

An Auxiliary Register of 38 bits. This is used with the Accumulator to form a double-length register.

A Sequence Control Register of 14 bits.

An Overflow Indicator of 1 bit (O.F.R.).

An Interrupt Memory Register of 11 bits.

An Interrupt Permit Register of 1 bit.

An Interrupt Mask Register of 8 bits.

Throughout this volume the digits in any register are numbered sequentially starting from the least significant digit which is numbered 1.

NUMBER REPRESENTATION WITHIN THE COMPUTER

The fixed-point number representation within the computer is such that numbers are held in the range $-1 \leqslant x < +1$. The two's complement notation is used for negative numbers, i.e. they are represented by words beginning with 1, on the basis that the negative number $(-x)$ is represented by $(2-|x|)$ thus:

1 0101 1000 0000 0000 0000 0000 0000 0000 0000 00

$$\text{this is } 1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{32} = 1\frac{11}{32}$$

$$\text{which is } (2-\frac{21}{32}) \text{ so the word represents } \frac{-21}{32}$$

The largest possible fraction is $1 - 2^{-38}$

If the number is an integer then its binary representation is stored at the least significant end of the register, but if it is a fraction it is stored at the most significant end unless it is moved down the register by scaling.

The largest possible integer is 274 877 906 943. To avoid ambiguity, the fixed-point numbers below will be fractions.

The standard floating-point form is $x = a.2^b$ such that $-1 \leqslant a \leqslant -\frac{1}{2}$ or $a = 0$ or $\frac{1}{2} \leqslant a < 1$ and $-256 \leqslant b \leqslant 255$. The mantissa (a) of the number is represented by bits 10 to 39 inclusive of the word and the exponent (b) is represented by bits 1 to 9 inclusive.

Floating-point round-off is obtained by forcing the last of the 30 digits, representing the mantissa after standardization, to be 1 in all cases except where the number is exactly representable in 30 bits of signed binary. The maximum error after any operation is therefore less than 1 in the 2^{-29} position of the mantissa. This means that a floating-point number can be represented to an accuracy of 8 significant figures.

The instruction code is divided into eight groups. The functions in groups 0 to 3 are termed **Basic Functions** and these, together with the Transfer Functions of Group 4, have in common that the N digits always specify the address of a location. In some cases (functions 00, 01 and 06), the content of that location does not enter into the operation. The uses of the N digits in the functions of groups 5 to 7 are given in the appropriate sections below. The times taken to perform all functions are given in 1.2.1.

GROUP 0

Note: None of the functions in this group affects the content of any store location (Register). In functions 0-4, A is the content of the Accumulator and R the content of the Register.

Function 00 (AR) *The contents of A and R are unchanged.*

This function is especially useful in conjunction with the B digit (see B-Line Modification).

Function 01 (MR) *A is deleted and replaced by Minus A.*

Function 02 (IR) *A is deleted and replaced by R Incremented by* 1.

The existing C(A) is deleted and replaced by a number equivalent to $C(N) + 2^{-38}$.

Function 03 (CR) *Collate or logical and.*

A new word is formed which has a 1 in each position in which there are 1's in both C(A) and C(N), and 0's elsewhere. C(A) is then deleted and replaced by this new word.

Example 1

The collate function may be used when it is desired to extract the wanted number from a location in which two numbers have been packed. If, for example, two positive integers 127 (1111111) and 341 (101010101) have been stored together in location 300, the contents of location 300 will be as follows:

```
0 0000 0000 0000 1111 1110 0000 0000 0101 0101 01
```

i.e. $127 \times 2^{-19} + 341 \times 2^{-38}$. If the integer 524287 is stored in location 276, then the following instructions suffice to get the right-hand number into the accumulator on its own:

Address	F1	N1	B	F2	N2
740	30	300	:	03	276

For the effect is to collate

```
0 0000 0000 0000 1111 1110 0000 0000 0101 0101 01
```

with the **collating constant**

```
0 0000 0000 0000 0000 0001 1111 1111 1111 1111 11
```

and thus obtain

```
0 0000 0000 0000 0000 0000 0000 0000 0101 0101 01
```

Function 04 (PR) *A is deleted and replaced by A Plus R.*

C(N) is added to C(A), which is deleted and replaced by this sum.

Function 05 (SR) *R is Subtracted from A which is deleted and replaced by this difference.*

Function 06 (ZR) *A is deleted and replaced by Zero.*

Function 07 (DR) *A is subtracted from R, A is deleted and replaced by the Difference.*

The result is the negative of that produced by function 05.

GROUPS 1 to 3

In groups 1 to 3 the operations are similar to those described above, but whereas in Group 0 the result is placed in the accumulator and the store remains unaffected, the arrangements in Groups 1 to 3 vary. In Group 1 the result is placed in the accumulator and the original content of the accumulator is placed in the store. In Group 2 the result is placed in the store and the accumulator remains unaffected. In Group 3 the result is placed in the store and the original content of the store is placed in the accumulator (see table below).

The contents of the accumulator and specified location are denoted by a and n.

GROUP 0				GROUP 2			
		Contents after the function has been performed				Contents after the function has been performed	
Code	Mnemonic	C(A)	C(N)	Code	Mnemonic	C(A)	C(N)
00	AR	a	n	20	AA	a	a
01	MR	−a	n	21	AM	a	−a
02	IR	n+1	n	22	AI	a	n+1
03	CR	a & n	n	23	AC	a	a & n
04	PR	a+n	n	24	AP	a	a+n
05	SR	a−n	n	25	AS	a	a−n
06	ZR	0	n	26	AZ	a	0
07	DR	n−a	n	27	AD	a	n−a
GROUP 1				**GROUP 3**			
		Contents after the function has been performed				Contents after the function has been performed	
Code	Mnemonic	C(A)	C(N)	Code	Mnemonic	C(A)	C(N)
10	RA	n	a	30	RR	n	n
11	NA	−n	a	31	RN	n	−n
12	IA	n+1	a	32	RI	n	n+1
13	CA	a & n	a	33	RC	n	a & n
14	PA	a+n	a	34	RP	n	a+n
15	SA	a−n	a	35	RS	n	a−n
16	ZA	0	a	36	RZ	n	0
17	DA	n−a	a	37	RD	n	n−a

Arithmetic operations (i.e. 04, 05, 07) only refer to fixed-point operations (see 1.2.1).

GROUP 4—JUMP INSTRUCTIONS

The 503 normally obeys instructions sequentially but this sequence can be changed by the use of Group 4 instructions. Jump or, as they are sometimes called, transfer instructions, are inserted in a program wherever it is desired that the computer should obey instructions in a sequence different from that in which they have been stored. The address of the next instruction to be obeyed is stored in the Sequence Control Register (S.C.R.).

Function 40 (JUF) *Jump Unconditionally, to First instruction*

This causes the computer to break normal sequence, carry out the first instruction in location N, and then continue in normal sequence from that new point until a further transfer instruction is obeyed.

Function 41 (JNF) *Jump if C(A) is Negative to First instruction*

If the sign digit in the computer is a 1 (content negative) when this instruction is obeyed, the action is as for function 40. But if the sign digit is 0 (content zero or positive), the instruction is interpreted as 'do nothing'.

Example 2

Suppose that in some iterative process, certain steps of a calculation must be performed 6 times. This could be organised as in the program below in which the **entry point** is the first instruction in location 10.

Address	Instructions					
	F1	N1	B	F2	N2	
9					−5	This is the way in which the number 5×2^{-38} is expressed on a program sheet.
10	30	9	:	20	8	Place −5 in location 8.
11						
	The instructions to be obeyed 6 times.					
15						
16	32	8	:	41	11	Increase the content of location 8 by 1, and repeat the set of instructions if necessary.
17	Next part of program.					

When the instruction 32 8 is obeyed for the first time, the number which comes into the accumulator is −5. On the second occasion it is −4, then −3, −2, −1 and finally after the sixth iteration it is 0.

In the above program 'a count is kept in location 8' or 'location 8 is used as a count location'. The number placed in a count location at the start of such a process is called a 'count constant'. The action of placing a count constant in a count location is termed 'setting the count'.

Function 42 (JZF) *Jump if C(A) is Zero, to the First instruction*

If the content of the accumulator is zero when the computer obeys this instruction, the action is as for function 40. But if the content is not zero, the instruction is interpreted as 'do nothing'.

Function 43 (JOF) *Jump, if the Fixed-point Overflow indication is set, and clear the overflow indicator.*

The overflow indicator is tested, and if found to be in the set condition, the action is as for function 40, coupled with the fact that the overflow indicator is cleared. But if the overflow indicator is found to be in the clear condition, the instruction has the same effect as 'do nothing'. (See later for a list of functions which can cause fixed-point overflow.)

If it is desired to use the overflow indicator to test whether some function or group of functions has caused overflow, it is necessary to ensure that the overflow indicator is clear before they take place. This may require the insertion of an instruction whose only effect is to clear the overflow indicator (if it is set). The instruction 43 (N+1) placed in the second position of location N has this effect.

Functions 44, 45, 46 and 47

These functions are identical in operation to 40, 41, 42 and 43 respectively, except that where a transfer takes place it is to the second instruction in the specified location instead of the first. Their mnemonic codes are the same as those for 40 to 43 respectively except that the F in the code is replaced by an S, e.g. 44 (JUS).

GROUP 5—MULTIPLICATION, DIVISION AND SHIFT FUNCTIONS

When two fractions of equal length (e.g. + .23 and − .47) are multiplied, the answer, in general, contains twice as many digits as either (− .1081). The scaling factor is determined in the same way as the decimal point is determined in a long hand multiplication. Similarly, if two computer words each of 38 fractional digits and one sign digit are multiplied, the full product has 76 fractional digits and one sign digit. Provision is made in the 503 to 'extend' the accumulator in this and certain other cases to accommodate one such **double-length** number. This is held in such a way that the sign digit and the next 38 digits are in the accumulator itself, while the remaining 38 digits fill the extension, which is termed the Auxiliary Register (A.R.). Group 5 functions do not affect the store.

In the case of functions 52, 53 and 56 (multiply and divide), the N digits specify the address of a location in the store. In function 57 (read A.R.), the N digits are not used. In functions 50, 51, 54 and 55 (shifts), the seven right-hand N digits specify a number $\underline{N}$: this $\underline{N}$ has therefore a range of from 0 to 127 inclusive, and if N=145 say then $\underline{N}$=17.

Function 50 (DLH) *Halve the Double-Length number* $\underline{N}$ *times*

The digits of the double-length number are shifted $\underline{N}$ places to the right, the $\underline{N}$ right-hand digits of the A.R. being lost. If the original number is positive or zero, $\underline{N}$ zeros are inserted at the left-hand end: if it is negative $\underline{N}$ ones are inserted.

Thus the sign of the number is maintained, and the effect is a true division by $2^{\underline{N}}$, subject to any error occurring through the loss of right-hand digits of the A.R.

Example 3

If the double-length numbers $\frac{1}{2}$ and $-\frac{1}{2}$

```
i.e.  0  1000  0000  ..............................................  0000  0000
and   1  1000  0000  ..............................................  0000  0000
                          are halved 4 times, the results are
      0  0000  1000  ..............................................  0000  0000
and   1  1111  1000  ..............................................  0000  0000
```

which are $\frac{1}{32}$ and $\frac{-1}{32}$ respectively.

Function 51 (SLH) *Halve Single-Length. Clear A.R.*

The digits in the accumulator are shifted N places to the right, the $\underline{N}$ right-hand digits being lost, and $\underline{N}$ zeros are inserted at the left. The A.R. has no part in this process: it is cleared as a separate operation. The sign digit is shifted along with the other 38 digits. 51 0 $0 \rightarrow B$

Function 52 (DLM) *Multiply with Double-Length product*

The existing C(A) is multiplied by C(N). C(A) and the existing content of the A.R. (which is **not** used in the function) are deleted, and replaced by the double-length product. Since the multiplier and multiplicand are both single length the result is always exact.

Function 53 (SLM) *Multiply with Single-Length rounded product. Clear A.R.*

The existing C(A) is multiplied by C(N), C(A) is deleted and the double-length product is formed, the quantity 2^{-39} is artificially added and the left-hand 39 digits of the result are placed in the accumulator. The A.R., whose original content is not used in the function, is cleared.

Example 4

If 9×2^{-20} and 3×2^{-20} are multiplied using the function 52, the result is 27×2^{-40}, held as a double-length number. 27 in binary is 11011, so the result appears thus:

```
                   Accumulator                                  A.R.
0  ..........................  0001  10     11  0000  ........................  0000
```

But if function 53 is used the process is, in detail, as follows:

(a) the double-length product is formed.

(b) the quantity 2^{-39} is artificially added.

(c) the left-hand 39 digits of the result are placed in the accumulator, the other 38 digits are lost, and the A.R. is cleared.

The arithmetic may be represented thus:

```
                    Accumulator                               A.R.
(a) 0 ........................ 0001  10    11  0000 ..................... 0000
+                                          1
(b) 0 ........................ 0001  11    01  0000 ..................... 0000
(c) 0 ........................ 0001  11    00  0000 ..................... 0000
```

which is 7×2^{-38}, the correct approximation.

Function 54 (DLD) *Double the Double-Length number* $\underline{N}$ *times*

In this function, the digits of the double-length number are moved $\underline{N}$ places to the left, $\underline{N}$ zeros are inserted at the right-hand end of the A.R. and the $\underline{N}$ left-hand digits are lost. Unless overflow takes place the effect is multiplication by 2^{N}.

Hence if $\underline{N} \geqslant 38$, the A.R. is clear after this function.

Function 55 (SLD) *Clear A.R. Double the Single-Length C(A)* $\underline{N}$ *times.*

The digits in the accumulator are moved $\underline{N}$ places to the left, $\underline{N}$ zeros are inserted at the right-hand end, and the $\underline{N}$ left-hand digits are lost. Unless overflow takes place, the effect is multiplication by 2^{N}.

The A.R. has no part in this process: it is cleared as a separate action.

Function 56 (DIV) *Divide. Clear A.R.*

Division is the inverse of multiplication, in that as $a \times b = ab$ so $ab/_b = a$. Just as it is useful to have a double-length 'ab' product in multiplication, so it is useful to have a double-length 'ab' (numerator, or dividend) in division.

Function 56 therefore consists of the division of the double-length number by C(N). C(A) is deleted and replaced by the single-length, unrounded quotient. The A.R. is cleared by the process.

When the numerator can be adequately expressed by a single-length number in the accumulator, care must be taken to ensure that the A.R. is clear before division takes place.

The accuracy of the division process

This concerns the accuracy of the result produced by the computer when the true quotient lies within the range -1 to $+1$ inclusive.

(a) Since the result is not rounded, when the true quotient is not exactly expressible as a 39 digit computer number, the result may be 2^{-38} less than that which would be obtained by using a process containing a round-off stage.

(b) If a and b are both positive, and $b > a$, and if the quotient is exactly expressible as a 39-digit computer word,

then for

a/b, $-a/b$, $0/b$: the computer results are correct

but for

$a/-b$, $-a/-b$, $0/-b$: the computer results are 2^{-38} less than the arithmetically correct results.

(c) For the division $\pm a/\pm a$, $a > 0$, the effects of (b), combined with the fact that there is no computer representation of $+1$, cause the results shown below to be produced. Note also the result obtained for 0/0, and that the overflow indicator is set for this case, and for both cases in which the result of $\pm a/\pm a$ errs by more than 2^{-38}.

Division	*Computer Result*	*Overflow Indicator*
a/a	-1	Set
$-$a/a	-1	Not Set
a/$-$a	$1-2^{-38}$	Set
$-$a/$-$a	$1-2^{-38}$	Not Set
0/0		Set

Example 5

(a) $C(149)=2\times2^{-38}$, $C(150)=1\times2^{-38}$
$C(151)=a\times2^{-38}$, $C(152)=b\times2^{-38}$
To place $a/b\times2^{-38}$ in 153

Address	F1	N1	:	F2	N2	
		Instructions				
387	30	151	:	52	149	Form $2a\times2^{-76}$
388	56	152	:	04	150	Divide and form $\frac{(2a+1)}{b}\times2^{-38}$
389	50	1	:	20	153	Halve and write

This routine computes the correct answer in all cases, rounding towards zero when the answer is not an exact integer.

(b) $C(251)=a\times2^{-19}$, known to be positive
$C(252)=b\times2^{-19}$. To place $a/b\times2^{-19}$ in location 253

Address	F1	N1	B	F2	N2	
		Instructions				
487	30	251	:	51	19	Form $a\times2^{-38}$ and clear A.R.
488	56	252	:	20	253	Divide, and write result.

(c) $C(351)=a\times2^{-19}$, sign unknown, $C(352)=b\times2^{-19}$
State of A.R. is not known. To place $a/b\times2^{-19}$ in location 353

587	51	0	:	30	351	Clear A.R., replace $a\times2^{-19}$
588	50	19	:	56	352	Divide $a\times2^{-38}$ by $b\times2^{-19}$
589	20	353	:			Write result.

The instructions 51 0 and 55 0 have the sole effect 'Clear the A.R.'

Function 57 (RAR) *Read Auxiliary Register*

The existing C(A) is deleted by a word comprising a zero in the sign digit position and a copy of the 38 digits of the A.R. content, in the remaining positions.

The A.R. is not affected.

This provides, in effect, a fast double-length left shift of 38 places, subject to the following provisos:

(i) That the sign digit will be wrong if the number is negative.

(ii) That the A.R. is not cleared.

The second of these points is usually of little consequence: in fact, it may well be found to be an advantage. How the former is overcome is shown below.

Example 6

The following process is faster than would be the case if 54 38 were used:

a and b are integers of unknown sign, in locations 643 and 644. It is required to place the integer ab in location 645, where $| ab | < 2^{37}$.

Address	F1	N1	B	F2	N2	
		Instructions				
412	30	643	:	52	644	$ab\times2^{-76}$
413	57	0	:	54	1	Double signless $ab\times2^{-38}$
414	50	1	:	16	645	Store signed $ab\times2^{-38}$

Example 7

This emphasises the advantage of multiplying double-length and dividing, successively, whenever an expression of the form

$$\frac{a.c.e.\dots\dots}{b.d\ \dots\dots}\text{ is to be evaluated.}$$

Suppose that a, b and c are stored to scale p (where p may be of the form 2^{-r} or not), in locations 141 to 143. To evaluate ac/b and place it to scale p in location 144, we need:

Address	Instructions F1	N1	B	F2	N2	
614	30	141	:	52	143	$a\times p : ac\times p^2$
615	56	142	:	20	144	$ac/b\times p$:

This gives a clue to the method used to rescale after multiplication or before division when using scales not of the form $\times 2^{-r}$. For if we make b=1 the result is $ac\times p$; if c=1 the result is $(a/b)\times p$.

Example 8

To extract the positive square root of a fraction, x.

If the formula $y_{n+1} = \frac{1}{2}(y_n + \frac{x}{y_n})$

is used to generate a series of numbers y_i, it can be shown that successive values of y_i converge monotonically to the square root of x whatever initial value of y_i is taken.

In adapting this process to the 503, $y=1-2^{-38}$ is chosen, as this is the largest positive number which can be held, and must therefore be greater than or equal to the required root. Thus, until the best possible approximation to the root is found each new y_i is less than the previous one, and $y_{n+1} - y_n$ is negative.

Ideally, $y_{n+1} - y_n$ should eventually reach zero, but rounding errors may bring about a situation in which it oscillates between $\pm 2^{-38}$. The process is therefore repeated until some $y_{n+1} - y_n$ is zero or positive, at which stage the program 'exits with y_n in the accumulator'.

Program

It is assumed that x is in the accumulator, and that the A.R. is clear. The entry point is the first instruction in location 501.

Address	Instructions F1	N1	B	F2	N2	
501	41	501	:	42	507	If negative stop. If zero exit.
502	20	509	:	30	508	Write x
503	20	510	:	30	509	Write y_1. Replace x
504	56	510	:	04	510	Form $2y_{n-1}=(y_n + \frac{x}{y_n})$
505	51	1	:	15	510	Write y_{n+1}, form $y_{n+1} - y_n$
506	45	503	:	07	510	If negative, repeat: if positive exit with y_n
507	40	511	:	00	0	EXIT
508	37	8191	/	77	8191	This gives a pattern of 0 followed by 38 ones which is $1-2^{-38}$
509						x
510						y_n
511	Next part of program					

GROUP 6—AUTOMATIC FLOATING-POINT ARITHMETIC

In functions 60 to 64 inclusive the computer treats both a and n as standard floating-point numbers and produces a standard floating-point result, which replaces the old content of the accumulator.

In functions 60 to 64 inclusive the content of the auxiliary register is cleared but the store is not affected.

For the accuracy of floating-point operations see 'NUMBER REPRESENTATION WITHIN THE COMPUTER'.

Function 60 (FPA) *Add in Floating-Point mode. Clear A.R.*

C(N) is added to C(A) which is deleted and replaced by the sum. The A.R. is cleared by the process.

Function 61 (FPS) *Subtract in Floating-Point mode. Clear A.R.*

C(N) is subtracted from C(A) which is deleted and replaced by the difference. The A.R. is cleared by the process.

Function 62 (FPN) *Negate and add in Floating-Point mode. Clear A.R.*

C(A) is subtracted from C(N), the original C(A) is deleted and replaced by the difference. The result is the negative of that produced by function 61. The A.R. is cleared by the process.

Function 63 (FPM) *Multiply in Floating-Point mode. Clear A.R.*

C(A) is multiplied by C(N), the original C(A) is deleted and replaced by the product. The A.R. is cleared by the process.

Function 64 (FPD) *Divide in Floating-Point mode. Clear A.R.*

C(A) is divided by C(N), the original C(A) is deleted and replaced by the quotient. The A.R. is cleared by the process.

Function 65 (SDZ) *Standardize. Clear A.R.*

The 39-bit integer in the accumulator is converted to standard floating-point mode. The A.R. is cleared by the process.

Conversion

The only permitted form of the 65 function to convert an integer to floating-point representation is 65 4096.

In obeying the instruction 65 4096 the computer treats the existing content of the accumulator as a fixed-point integer to scale $\times 2^{-38}$, and converts this to standard floating-point form.

Example

0 00000000000000000000000000000000001111

which is the fixed-point representation of the integer 15, is converted to

0 1111000000000000000000000000 100000100

$$\text{in which } a = \frac{15}{16} \text{ and } (b + 256) = 260$$

$$\text{so that } a \times 2^b = \frac{15}{16} \times 2^4 = 15$$

Function 66 (LOD) *Load registers and transfer control*

This instruction is used to exit from an interrupt program (see 1.2.3 and 1.2.4).

The effect of the instruction 66 L is as follows:

(a) Set the modifier register from location L and set the modifier indicator.

(b) Restore Acc. from location L+1.

(c) Restore A.R. from digit positions 1 to 38 of location L+2.

(d) Restore the fixed-point overflow register (OFR) from position 39 of location L+3, i.e. if digit 39 of location L+3 is 1 set OFR.

(e) Set S.C.R. to the value contained in positions 1 to 14 of location L+3 (i.e. cause a transfer of control) and remove the inhibition on interrupts by setting the permit bit to 1. (For further details see 1.2.4).

Function 67 (MOD) *Modify the next instruction by adding to it the least significant* 19 *digits of n.*

The instruction is modified just before it is obeyed, and its stored form is unaltered. (See B-modification.)

GROUP 7

The instructions in group 7, apart from 73, deal with the control of peripheral equipment. Details of these instructions are given in the sections of volume 1 which deal with the peripheral devices. The PCP (Peripheral Control Program) gives the macro instructions for the peripheral devices. A general description of the group 7 functions follows:

Function 70 (WGR) *Read from the Word Generator to the Accumulator*

The existing C(A) is deleted, and replaced by a copy of the word set on the word generator, (see 3.1.2). The N digits of the instruction are not used.

Function 71 *Read one character from a specified input device to the Accumulator*

Output N to control console for tape reader or typewriter selection; mix 8 bits, some of which may always be zero, from control console with bits 1 to 8 of the accumulator.

(TR1)	For $0 \leq N \leq 2047$	channel 1 is the selected input channel.
(TR2)	For $2048 \leq N \leq 4095$	channel 2 is the selected input channel.
(DTR)	For $4096 \leq N$	channel 3 is the selected input channel.

Unless the reader SELECT button is depressed (see 1.2.2 or 3.1.2), channel 1 is tape reader 1, channel 2 is tape reader 2 and channel 3 is the control typewriter.

After the character has been read, the tape reader drive mechanism operates, and the tape is moved forward to bring the next character into the reading position.

Function 72 *Transfer a single word from the Accumulator to a specified peripheral device*

Output N to peripheral controllers; output a single word from the accumulator to the peripheral device. This function is used to transfer a single word from the accumulator to the peripheral device. For an example see 1.4.1.

Function 72 has a special application when $N \leq 511$. It is then used to select which interrupt lines may be operative in a particular program (see 1.2.3 and 1.2.4).

Function 73 (SCR) *Write the address of this instruction, i.e. store S.C.R.*

Write the address of the present instruction in positions 1 to 13 of location N of the store. The remaining positions of N being zero.

This instruction is generally used in conjunction with the B-digit for entering subroutines (see later).

Function 74 *Output one character on a specified output device*

Output N to the Control Station for selection purposes, then output N to Control Station as information.

(TP1)	For $0 \leq N \leq 2047$	channel 1 is the selected output channel.
(TP2)	For $2048 \leq N \leq 4095$	channel 2 is the selected output channel.
(DTW)	For $4096 \leq N$	channel 3 is the selected output channel.

Unless the punch SELECT button is depressed (see 1.2.2 or 3.1.2), channel 1 is tape punch 1, channel 2 is tape punch 2 and channel 3 is the output writer.

When the necessary signals have been sent to the punch, the computer is free to proceed with other activity.

Function 75 *Transfer a single word from a specified peripheral device to the Accumulator*

Output N to peripheral controllers. Input a single word to Acc. from a peripheral device. This function is used to transfer a single word from a peripheral device to the accumulator. For an example see 1.4.1.

Function 76 *Output N to peripheral controllers*

Input a number of control bits from a specified peripheral device to the Acc. Prepare for autonomous transfer. (See 1.3.1). This instruction is used in conjunction with the 77 instruction for performing block transfers.

The effect of the instruction 76 N is as follows:

(1) Place the 76 address code in location 8176 to scale 2^{-38}.

(2) Input a control word from the specified device to Acc. The control word contains digits indicating the state of the device. The precise significance of the digits depends on the type of device. If the specified controller does not exist an error interrupt occurs.

(3) The N digits of the 76 instruction specify the device and the type of transfer required:

m.s.	Digits 13 to 10	Digits 9 to 4	Digits 3 to 1	l.s.
	Class of device	Number of device	Operation	

(For further details see 1.3.1)

Function 77 *Execute the operation specified by the last* 76 *instruction. If a transfer was specified use locations N onwards in the store*

The effect of the instruction 77 N is as follows:

(1) Read the 76 address code from location 8176. Use this code to select a device and test if it is busy. If the device is busy then re-read the 76 address code and repeat the testing process. If a non-existant controller is selected then cause an error interrupt.

(2) Set in the location associated with the peripheral controller a pair of counts, i.e. 77 8192-T : 77 N-1.

(3) Clear location 8176 after having stored its contents internally.

(4) Tag the store area associated with the transfer. If a location is found already tagged wait until the tag disappears. The A.R. is used in this process and is not clear on exit.

After step 4 the program starts to obey the next instruction. The peripheral device has been activated by the completion of step 4 and when the device is ready to transfer a word the following operations take place:

(5) The central processor completes its current instruction.

(6) Both the counts, set in step 2 above, are incremented by 1 and a word is transferred to or from the location specified by the lower count. At the same time the tag digit is cleared.

(7) The upper count is tested. If the count is non-negative then the transfer has been completed.

(For further details see 1.3.1).

FUNCTIONS WHICH CAN CAUSE FIXED-POINT OVERFLOW

Title	*Function*	*Circumstances causing Overflow*
Minus or Negate	01, 11, 21, 31	In the case where the number to be negated is -1.
Increment	02, 12, 22, 32	When C(N) is $1-2^{-38}$.
Plus	04, 14, 24, 34	Whenever the sum is outside the range -1 to $1-2^{-38}$ inclusive.
Subtract	05, 15, 27, 37	Whenever the difference is outside the range -1 to $1-2^{-38}$ inclusive.
Difference	07, 17, 25, 35	
Multiply	52, 53	In the case of -1 times -1 only.
Double	54, 55	Whenever the correct result lies outside the range -1 to $1-2^{-38}$ inclusive.
Divide	56	Whenever the modulus of the numerator (dividend) is greater than that of the denominator (divisor) and for a/a when the computer result is -1, a/$-$a when the result is $1-2^{-38}$ and for 0/0 where the result is 0

B-LINE MODIFICATION

Instances frequently occur in which the computer is required to carry out, several times, an action which is much the same at each repetition, but changes in some small detail each time it is performed. An elementary example is the addition of, say, 100 numbers stored in locations 100 to 199; after clearing the accumulator the functions to be performed are 'add C(100)', 'add C(101)', 'add C(102)' . . . . add 'C(199)'. The action here is always 'add', the change is an increase in the address of the addend.

Since instructions are stored in the computer in a numerical code, it is possible to cause a number to be added to an instruction and thereby change or **modify** it, before it is obeyed. This could readily be done by using such a function as 24, specifying the location holding the instruction to be modified. Better still, in the addition problem considered above, would be function 22; for the addition of 2^{-38} to the content of a location is tantamount to an increase by 1 of the N in the second instruction therein.

But an automatic facility termed **B-line modification** which is an improvement on either of the above is provided, and is as follows:

(i) If the B digit between a pair of instructions is a 0, then no modification takes place, and each instruction is obeyed as stored.

(ii) If the B digit between the pair of instructions is a 1, then after the first instruction (F1 N1) has been obeyed normally, the second instruction (F2 N2) is modified by the addition of the 19 least significant bits of the (new) content of location N1 before being obeyed.

The modification takes place just before the second instruction is obeyed and does not affect the version of the instruction (F2 N2) held in the store.

If the first instruction of a pair is a 77 instruction then the presence of a B-line between the two instructions does not cause modification.

It is conventional to indicate B=1 by placing a / in the B column rather than by writing a 1.

A B-line can only appear between two instructions held in the same word. The instruction 67 N can be placed anywhere and can therefore be used to modify either the first or second instruction in a word. The effect of the instruction 67 N is similar to 00 N followed by a B-line.

There are no exceptions to the action of modification so that:

(1) If a transfer control instruction is followed by a B-line the next instruction actually obeyed is modified.

(2) If control is transferred to an instruction which is preceded by a B-line or by a 67 instruction then no modification will take place (unless the transfer instruction is followed by a B-line as in (1) above).

(3) The instruction 67 N followed by a B-line is obeyed as if only one of these were present.

(4) Sequences of 67 instructions, possibly mixed with B-lines, result in sequences of modification.

Example 10

If it is desired to have an instruction (F2 N2) modified by the content of location 173 we could write any of the following:

F1	N1	B	F2	N2
00	173	/	F2	N2
67	173	/	F2	N2
67	173	:	F2	N2
22	173	/	F2	N2

In the fourth instance, where C(173) is altered by function 22, the modification is by the new content.

Example 11

The following program adds together 100 items, namely, the contents of locations 100 to 199 inclusive, and exits with the sum in location 238. It is entered at the first instruction of location 240. (It is assumed that the sum is within capacity.)

Address	Instructions					
	F1	N1	B	F2	N2	
238						Count location and total store.
239					−100	Count constant
240	26	238	:	30	239	Set total to zero: replace −100
241	10	238	/	04	200	Set count: add an item
242	12	238	:	41	241	Count: test for end.
243	Next part of program.					

Note how the powerful 'exchange' functions 10 and 12 are used. They enable the total-to-date and the count to be switched between location 238 and the accumulator. Without such functions it would be necessary to allocate separate locations for the count and the total, and there would be more instructions in the loop.

SUBROUTINES

Suppose that during a lengthy calculation, it is necessary to find the square root of several numbers. The need would then arise for the computer to carry out the instructions of Example 8 several times, and a considerable saving in storage requirements could be achieved by arranging that the same set of instructions is used each time, rather than by having several similar sets, each to be used once.

This technique can be applied to any small section of program which is designed to fulfil a standard requirement, and such sections of program are termed **subroutines.** Many of the subroutines in the 503 Library are S.A.C. subroutines. The **link** for these S.A.C. subroutines is described in 2.1.2.

To appreciate the value of this in programming consider again the case in which several square roots must be found during a calculation. Provided the subroutine sqrt is in the store the S.A.C. instruction

SUBR, sqrt.

may be written as many times as is necessary, and be regarded as the equivalent of one instruction: 'form the square root of C(A)'.

This technique is employed even when a subroutine is used only once during a calculation. For, by doing so, the need to have different versions of each subroutine is avoided, and so is the clerical effort of copying the instructions into the middle of a new program.

CHAPTER 2
INTERRUPTS

On any but the basic 503 the interrupt facilities are under the control of PCP. This chapter is therefore, primarily intended for programmers who wish to use these facilities on the basic machine.

A description of the registers used and the different types of interrupt which can take place is found in sections 1.2.3, 1.2.4, and 1.2.5 of the Manual.

INTERRUPT REGISTERS

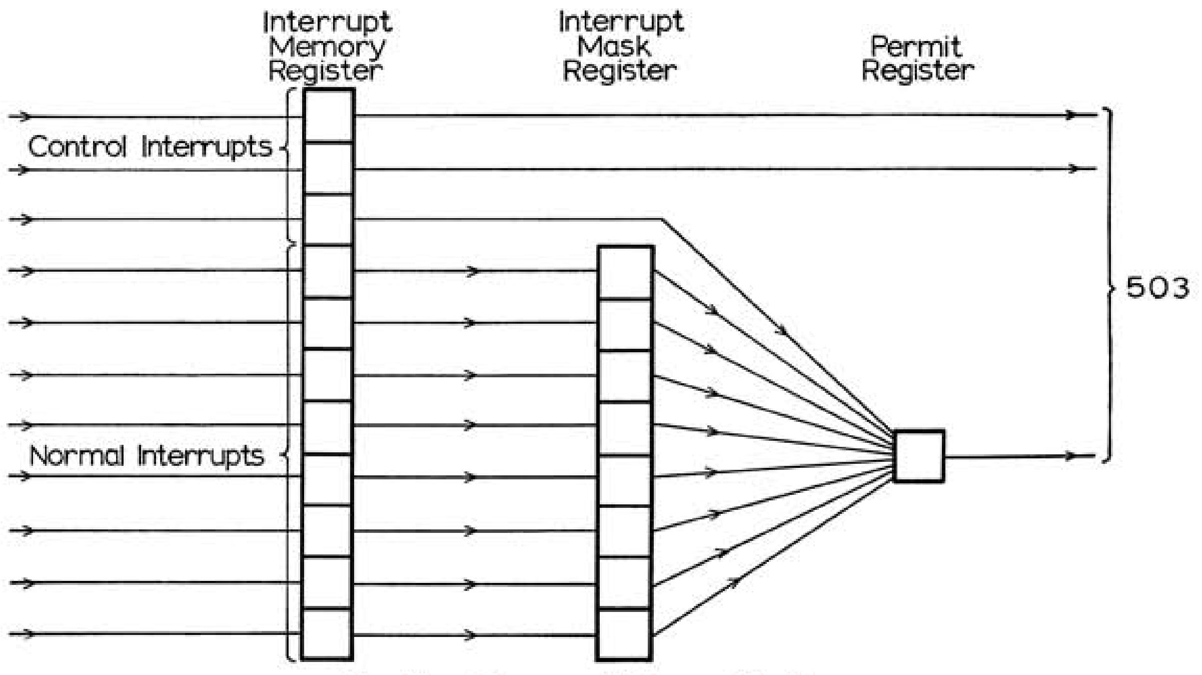
There are three registers of interest to the programmer associated with interrupts: the Interrupt Memory register (11 bits), the Interrupt Mask register (8 bits) and the Permit register (1 bit). (See functional diagram.)

There are two types of interrupt which can take place on the 503, Control interrupts and Normal interrupts. Three bits of the Interrupt Memory register are associated with Control interrupts and are set to one whenever a Control interrupt is requested. The other eight bits are associated with Normal interrupts and are set to one whenever a peripheral device indicates that it is free to accept further instructions. The Interrupt Memory register cannot be set by program. A one in any position of the register can only be cleared by an interrupt being accepted on that particular channel, or by the CLEAR or RESET buttons being pressed on the console. The register is also cleared if the machine is switched on with the Permit bit equal to one.

The Mask register is set by program to indicate which Normal interrupts are to be allowed and in what order of priority they are to be executed. Each bit of the register when set to one, allows interrupts on that channel (subject to the state of the Permit bit), but when set to zero inhibits them.

If the Permit register is set to zero, all interrupts with the exception of two control interrupts, are inhibited; if set to one, all interrupts are allowed provided the corresponding bits of the Mask register are also one.

Both the Mask register and Permit bit are also cleared when the CLEAR or RESET button on the console is pressed.



Functional Diagram of Interrupt Registers

TYPES OF INTERRUPT AND THE REQUIRED STATE OF THE REGISTERS

Control Interrupts

The three control interrupts occur due to machine or programming errors or due to the action of the operators. In order of priority they are:

1. Error interrupt

This takes the highest priority of all the interrupts and is not affected by the state of the Permit bit. It occurs automatically when one of the following error conditions arise: malfunction of the central processor or of the peripheral devices, faults in programming, or faults in manual operation, i.e.

(a) impermissible reference to the Reserved Area.

(b) unavailable peripheral device.

(c) main store parity.

(d) floating point overflow.

(e) special device interrupt—optional.

Advance warning of an error interrupt by a peripheral may be obtained by examining the control word of the device with a 76 instruction. It is then the option of the programmer either to take action to avoid the interrupt, or to ignore it.

2. Initial Instructions interrupt

This takes place when the INITIAL INSTRUCTIONS button on the control console is depressed. It occurs irrespective of the state of the Permit bit.

3. Manual interrupt

This takes place when the MESSAGE button on the control console is depressed, provided that the Permit bit is one. It cannot occur if the Permit bit is zero.

Normal Interrupts

The eight Normal interrupts are under the control of the programmer, to enable him to use the computer efficiently. The method used within a single program is to make the input and output routines run concurrently with the calculation part of the program. When there is insufficient calculation in one program to fill the time taken for input and output (or vice versa), it is more efficient to run two or more programs together.

The Normal interrupt of second priority is allocated to the paper tape readers; and the third to the paper tape punches and digital plotter. Non-basic machines with peripheral equipment capable of performing an A.D.T., have this equipment connected to the fourth position. (The other five channels can be connected up to other equipment as an optional extra.)

When these devices are free to accept further instructions a one is set in the corresponding positions of the Interrupt Memory register. The programmer can choose his own priorities for all Normal interrupts by placing the desired setting in the Mask register and putting a one in the Permit bit.

An interrupt can only be accepted when an instruction is about to be obeyed. This is true even when a B-modified instruction is about to take place. (i.e. the Interrupt Memory register is examined in order of priority each time an instruction is about to be obeyed).

A Control interrupt occurs as soon as a one is found in the Interrupt Memory register, with the exception of a Manual interrupt, when the Permit bit must also be present before the interrupt can occur. A Normal interrupt takes place as soon as a one is found in the Memory register, provided that a one is found in the corresponding bit of the Mask register and in the Permit bit.

If two interrupts are ready to be accepted at the same time, the one accepted is the interrupt of highest priority. This is the one with the most significant bit in the Interrupt Memory register.

PROCEDURE TAKEN ON ACCEPTANCE OF AN INTERRUPT BY THE 503

As soon as an interrupt is accepted the Permit bit is set to zero and the appropriate bit in the Interrupt Memory register is also made zero. The Permit bit is cleared to prevent further Normal or Manual interrupts before the interrupt in question has been dealt with.

The Permit bit is unaffected in the case of Initial Instructions interrupt in order that it may still be possible to interrupt manually should the need occur.

Each interrupt channel has associated with it, a single location called the Interrupt Location. This is examined on acceptance of the interrupt to determine the action to be taken. The Interrupt Locations are held in the Reserved Area, with the exception of that for the Initial Instructions which is location 0 and which can be considered to hold zero.

The Interrupt Location must contain:

	J	B		L
39.......34	33...............21	20	19.......14	13................1

(a) In digit positions 1 to 13 an address L which is the address of a block of four consecutive locations in which the contents of certain registers are to be stored when the interrupt takes place.

(b) In digit positions 21 to 33 the address J to which control is to be transferred when an interrupt takes place.

(c) In digit position 20, B, a bit which indicates whether control is to be transferred to the first or second instruction in the specified location. If B = 0 control goes to the first instruction of the location, but if B = 1 then the second.

The interrupt locations are 8166 to 8175 in the Reserved Area. 8166-8173 are associated with Normal interrupts, 8174 with Manual interrupt and 8175 with Error interrupt, i.e. the higher the address of the interrupt location the higher the priority.

Below is a summary table in order of priority of interrupts:

			After interrupt accepted	
Interrupt	**Mask Bit**	**Controlled by Permit Bit**	**Clears Permit Bit**	**Reads interrupt location**
Error	none	no	yes	8175
Initial Instructions	none	no	no	0
Manual	none	yes	yes	8174
Normal 1	8	yes	yes	8173
Normal 2	7	yes	yes	8172
Normal 3	6	yes	yes	8171
Normal 4	5	yes	yes	8170
Normal 5	4	yes	yes	8169
Normal 6	3	yes	yes	8168
Normal 7	2	yes	yes	8167
Normal 8	1	yes	yes	8166

After the computer has examined the interrupt location the following information is stored in locations L to (L+3):

L The content of the modifier register.

L+1 The content of the accumulator.

L+2 The content of the A.R. in digit positions 1 to 38. Bit 39 is made zero.

L+3 The content of the S.C.R. is stored in digit positions 1 to 14. The value stored is that associated with the instruction about to be obeyed. Digit position 39 holds the content of O.F.R. (1 if set, zero if not) and bits 34 to 38 are used for Error interrupt. A presence of a 1 in these bits indicates which error caused the interrupt.

bit 34. An attempted impermissible reference to the Reserved Area. The instruction containing the reference is not obeyed.

bit 35. Error due to an unavailable peripheral device (see below). The instruction causing the interrupt is not obeyed.

bit 36. A parity error in the main store. The instruction or autonomous transfer during which the error occurs is completed before error interrupt takes place.

bit 37. Spare. This class is provided in case devices requiring immediate interrupt are connected to the 503 at some later date. It is, however, set to 1 whenever the power is turned off.

bit 38. Floating point overflow. The instruction causing overflow is completed before error interrupt takes place.

After storing the required information in locations L to L+3, the S.C.R. is set to the value (J) given in bits 20 to 33 of the interrupt location thus transferring control to the required instruction.

In the case of Initial Instructions interrupt, pressure on the INITIAL INSTRUCTIONS button on the console, results in a transfer of control to location 0 without any actual storage of registers.

THE 72 AND 66 INSTRUCTIONS

The Mask register and Permit bit are set by one instruction 72 N. (where $N < 512$).

(Note: if $N \geqslant 512$ the computer acts on the peripherals).

The Permit bit is considered as occupying bit position 9. Thus $n \geqslant 256$ puts one in the Permit bit and $N < 256$ puts zero in the Permit bit.

e.g. 72 401
(00 . . . . . 01 1001 0001)

results in a one in the Permit bit and the pattern 1001 0001 in the Mask register with the most significant bit corresponding to that of highest priority.

i.e. if $N < 256$ Permit bit $= 0$, Mask register $= N$
if $N \geqslant 256$ Permit bit $= 1$, Mask register $= N-256$

In order to restore the registers and return to the position in the program before an interrupt occurred, the instruction

66 L

should be used. (L being the first location of the stored registers).

N.B. The Permit bit is set to one.

The sequence of operations is:

(a) Set the modifier register from location L and set the modifier indicator.

(b) Restore Acc. from location L+1.

(c) Restore A.R. from digit positions 1 to 38 of location L+2.

(d) Restore O.F.R. from position 39 of location L+3.
i.e. if bit 39 = 1 then set O.F.R.

(e) Set S.C.R. to the value contained in bits 1 to 14 of location L+3 (i.e. cause a transfer of control) and remove the overall inhibition by setting the Permit bit equal to one.

INTERRUPT FACILITIES IMPLEMENTING RAP

RAP provides the principal method of manual control for the 503 and assists the operator by displaying a message when an Error interrupt occurs and also when a Manual interrupt is accepted. It will then carry out instructions typed by the operator for continuation or otherwise.

RAP assists the programmer on the occurrence of an Error interrupt by making a search for an error routine to facilitate diagnosis of the error which caused the interrupt. It also provides a method of directing the course of a program after a Normal interrupt without unprotecting the Reserved Area. This is done by means of a SWITCH table held in the RAP workspace in which the programmer can set the destinations to which he requires the program to jump when a particular interrupt occurs. RAP arranges that the registers stored on interrupt are also placed in this SWITCH table so that return to the place where the program was interrupted is effected easily while the Reserved Area is still protected.

When an Error interrupt occurs, RAP displays

ERRINT n

where n is either 1, 2, 3, 4 or 5 corresponding to a bit in position 38, 37, 36, 35 or 34 respectively of location L+3.

RAP then searches for a program named ERRINT. If no such program exists, the message NOPROG. is displayed, otherwise control is transferred to it. This would usually be a short program to facilitate further diagnosis of the error.

When the MESSAGE button is depressed, a question mark is displayed to indicate a Manual interrupt is accepted and RAP awaits a message.

Continuing after a Control interrupt

In order to continue a program after an Error interrupt, type the message

CONT ; ERRINT.

The last program to be interrupted is continued from the point of interruption, even if a tape was being input when the interrupt occurred.

In the case of a Manual interrupt, the program can be continued from the point at which it was left by typing

CONT.

It is not possible to continue an interrupted program after an Initial Instructions interrupt since no registers are stored.

Settings of interrupt locations in RAP

Certain standard settings are placed in the interrupt locations in order that the destination of interrupts from Tape-Readers, Punches and digital plotter, and Peripherals may be controlled by a main store program when the Reserved Area is protected.

A SWITCH table is provided with 5 locations for each of the three interrupt channels mentioned above, outside the reserved area, in the RAP workspace.

The interrupt locations contain the following:

```
8172) 00 INT TR  : 00 (INT TR  +1)  tape readers
8171) 00 INT PUN : 00 (INT PUN+1)  tape punches and digital plotter
8170) 00 INT PER : 00 (INT PER +1)  peripherals
```

where the mnemonic names refer to the following absolute locations which must be used until SAP issue 3 makes provision for them.

name	*location*
INT TR	7887
INT PUN	7892
INT PER	7897

Thus when a tape reader interrupt occurs, the contents of the registers are stored in (INT TR+1), (INT TR+2), (INT TR+3), (INT TR+4), and control is transferred to the first instruction in location INT TR.

The programmer should store in INT TR the location to which he wishes to transfer control. Hence, to cause a reader interrupt to enter a main program block called TAPE the following SAC instructions are required:

```
30 < 40 TAPE : >
20    INT TR
```

The standard settings of INT TR, INT PUN, INT PER are such that interrupts occurring on these devices are ignored. These settings are only preserved if the user resets the appropriate location as follows:

```
30 < 00 8172/66 0 >
20    INT TR
```

IMASK

Since the subroutines of RAP affect the mask register and permit bit and there is no instruction for reading from the mask register, a copy of the content of the mask register must be kept at all times. A location is reserved in the workspace of RAP for this purpose and may be referred to mnenomically as "IMASK". On exit from RAP subroutines the mask register is reset in accordance with the contents of IMASK.

Whenever RAP is read into the store the contents of IMASK are set to a standard value which in the case of RAP issue 2 is zero. IMASK and the mask register are reset to this standard value each time a program is entered.

N.B. IMASK must be referred to by using the absolute value 7886 until SAP issue 3 makes provision for mnemonic reference to it.

To program interrupts when using RAP (issue 2)

The required jumps must be stored in the location INT TR, INT PUN, INT PER. The mask and permit registers must be set with a 72 instruction and a copy of the mask placed in IMASK. If there is a possibility that the program is to be run at the same time as another, care should be taken to ensure that the setting of the remaining bits of the mask register is preserved and that IMASK contains a record of this setting.

A. To allow a tape reader interrupt the following procedure is recommended:

	IMASK	Permit bit	Mask Register
72 0	. . . . 000 XXXX XXXX	0	0000 0000
30 < +191 >			
23 IMASK	. . . . 000 X0XX XXXX	0	0000 0000
30 < +64 >			
24 IMASK/	. . . . 000 X1XX XXXX		
72 256		1	X1XX XXXX

where X is original bit state.

B. To prevent Peripheral interrupts the following procedure is recommended:

	IMASK	Permit bit	Mask Register
72 0	. . . . 000 XXXX XXXX	0	0000 0000
30 < +239 >			
23 IMASK/	. . . . 000 XXX0 XXXX		
72 0		0	XXX0 XXXX

The permit bit is automatically cleared when an interrupt is accepted (with the exception of Initial Instructions interrupt) and exit from the interrupting routine, should normally release the overall inhibition thus set. To ensure that the main program is re-entered before other interrupts are permitted the 66 instruction and not the 72 256 instruction must be used, e.g.

```
67 IMASK
72 0
66 return
```

If a case occurs in which another selected interrupt is required during the interrupt routine, before returning to the main program, the overall inhibition must be released, thus:

```
67 IMASK
72 256
```

503 TECHNICAL MANUAL
VOLUME 2: PROGRAMMING INFORMATION
PART 1: PROGRAMMING SYSTEMS
SECTION 2: SYMBOLIC ASSEMBLY MARK I

January 1965
Issue 3

ELLIOTT

SCIENTIFIC COMPUTING DIVISION
Elliott Brothers (London) Limited
Elstree Way, Borehamwood, Hertfordshire, England

CONTENTS LIST

INTRODUCTORY NOTE

The following chapters assume a knowledge of the 503 machine code instructions. There are 64 orders and they are represented by pairs of octal numbers. Thus they fall into eight groups. Briefly groups 0-3 deal with fixed point addition and subtraction processes, group 4 are jump orders, group 5 perform fixed point multiplication and division, group 6 are floating point arithmetic functions and group 7 deal with input and output.

It is important that all users of the Symbolic Assembly Code make themselves familiar with these machine code instructions which are fully described in section 2.1.1.1. of the 503 Manual.

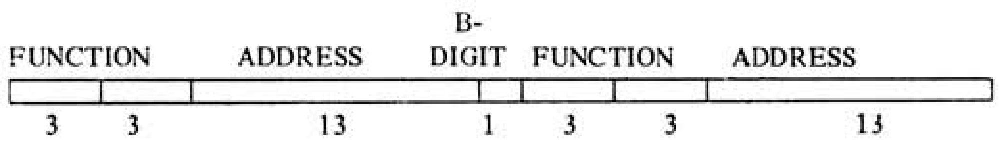
A basic knowledge of the Reserved Area Program is also assumed as this is the most fundamental program on the 503. See Section 2.2.1.

CHAPTER 1

FUNCTION OF THE SYSTEM

The Symbolic Assembly System is the normal medium for 503 Machine Code programming.

The 503 interprets each of the 39-bit information words in its store as a pair of instructions or as an item of data. The form of an instruction pair is:-



A SINGLE WORD

Although held in the computer in this form, each instruction is normally written on a separate line.

When the address part of each jump or data handling instruction is numerical (i.e. it refers to a fixed location in the store), the addresses are said to be 'absolute'. A program containing entirely absolute addresses must always use the same part of the store and a small modification may make it necessary to change many addresses.

In the Symbolic Assembly Code (SAC), the locations in the core store are referred to by mnemonic names chosen by the programmer. This has the effect of making the addresses relative, instead of absolute. Allocation of the absolute addresses is a final operation using information provided by the Reserved Area Program (RAP). The Symbolic Assembly Program (SAP) can translate a SAC program and plant it in a free part of the store, or produce an 'assembled' tape for subsequent use. Each assembled program has a name and can be run by typing this on the typewriter.

Thus, an assembled tape is in relocatable binary form, capable of being input under the control of RAP. After assembly into the store, or on input of the assembled tape, the program is held in absolute address form ready for running.

The advantages of SAC are as follows:-

1. Programming is simpler and less laborious.

2. Modifications to existing programs can be made easily.

3. Storage of several programs in the computer at the same time is easily organised, without causing overwriting etc.

4. The choice of mnemonic names can communicate the object of the program and how it is to be achieved.

Programming errors may be found at run time using SAPTEST. This can trace the course run by an assembled program and print out the contents of specified name-locations at specified stages.

CHAPTER 2

IDENTIFIERS, LABELS AND CONTROL NAMES

The mnemonic names mentioned in Chapter 1 must be more carefully defined.

2.1 **Identifiers** are used to distinguish by name each item of data used in a SAC program. They can take the form of any combination of letters, digits and spaces but must begin with a letter. The first six non-ignorable characters only are recognised. Thus no distinction would be made between Block 123, Block 1, Block 16.

An example of 5 identifiers is:-

a, data, a 5th CENTURY manuscript, AD 1962, pj 15 m 3L.

N.B. The characters paperthrow P, tab T, backspace B, runout R, space S, and erase E are ignored everywhere except in alphanumeric groups. The character newline L is used as a terminator.

2.2 Certain **Control Names** are used in SAC to define more closely the purpose for which each identifier is to be used. These names are composed of lower case letters which are underlined. Five basic control names are as follows:-

program	This introduces the one identifier which distinguishes and names the whole program. This statement must be the first made in any SAC program.
begin	The identifier introduced by this control name, distinguishes a certain section of program. In SAC all programs are written in easily handled parts called blocks. begin marks the start of such a block and instructions follow immediately after the identifier.
end	This control name does not introduce an identifier but merely marks the end of a block.
block	A list of all the identifiers marking each block included in the program must be defined after this control name. The statement is placed at the head of the program immediately following the initial statement declaring the program name. Each element in the list should be separated by a comma: separating new lines (L) are ignored.
data	This control name reserves a new storage location for each identifier which is named in the list following it. Locations reserved in this way are never re-used in the same program. Data identifiers may be introduced anywhere in a program provided that the introduction statement occurs before the end of the block in which they are needed.

Arrays can be declared under the latter control name, their length being indicated within round brackets after the selected data identifier, e.g. domino (12). This will cause 13 consecutive locations to be allocated, which can be referred to in the program as domino, domino+1, ...domino+12.

2.3 **Labels** are identifiers used as destinations of jump instructions. Introduction is effected by prefixing the label, followed by a closed bracket, to the instruction it is intended to identify. Each instruction may be given up to three labels.

e.g. L1) reallocation) repeat) 30 a

These labels may be used in the address part of any instruction within the blocks in which they are introduced. They refer only to the instruction which follows them: not the instruction pair. As the programmer is unlikely to know which location-half any labelled instruction will occupy when in the computer, SAP has been designed to supervise, and if necessary to correct, jump instructions with label identifiers in their address portions. Thus if the instruction labelled L1) . . . . . . lay in the second half of a location and a SAC program contained the instruction '43 L1', SAP would make this '47 L1'.

There is no provision for keeping fixed, instructions which are intended for use as parameters, and occur in the form of jumps to labelled points.

Labelling is an alternative way of booking dataspace. Although labels only identify single instructions, the effect of an instruction such as

30 reallocation

is to pick up the instruction-pair containing the instruction labelled 'reallocation'.

Thus, the occurrence of

```
L1) + 0
L2) + 0
```

in a SAC program causes two locations to be reserved within the program area, and these may be referred to as 'L1' and 'L2'.

2.4 Local and Global Identifiers

Labels and data identifiers introduced in the way outlined above have no meaning outside the blocks in which they are introduced, and are called LOCAL identifiers. GLOBAL identifiers require introduction between or before program-blocks and thus have the same meaning throughout a whole program.

The same local identifier can be used in several blocks without confusion, since it will be assigned a different meaning within each block. Block names are global, but data and label identifiers can be local, global or both. In blocks within which an identifier has both global and local references, the local reference is the one taken.

The same identifier can be used globally as a block name and locally as either a data or label identifier in any block of the program. The same identifier must never be used as both a data and label name in any one block.

2.5 Introduction of Global labels and data elements

Global labels are first introduced with the block identifiers, at the beginning of the program. Each block-identifier is followed, if necessary, by a list of the global labels which will be introduced within the block it identifies. The elements of this list are separated by commas, and the whole list is enclosed in round brackets.

Thus, an introduction of block identifiers might take the form

block silverbirch, fir (needle, cone, fibre), oak (timbers, surge);

so that it would be possible to refer to the labels needle, cone, fibre, timbers and surge in any block within the program.

Each global label must of course also appear as a prefix to the required instruction in the block named.

Block identifiers may be used as if they were labels attached to the first instructions of their blocks (but see restriction Paragraph 3.5).

Global data identifiers are defined under the control name data between or before the program blocks.

2.6 Relative identifiers

A relative identifier has the form A * B where 'B' is a blockname and 'A' is either a label or data identifier.

For example the instruction 40 A * B causes transfer to be made to that instruction in block 'B' which has been labelled 'A'. Note that 'A' must have been introduced as a global label at the head of the program.

Global label elements are referred to by their own identifiers, inside their own blocks, in the same way as local label and data elements. Outside their own blocks, global label elements are referred to by means of relative identifiers.

Global data elements may be associated with particular blocks by introducing them globally, but in relative identifier form.

The introduction

data D, D * B1, A(5), C,D * B3(4);

for example, causes 14 distinct locations of data space to be reserved at assembly time. Outside blocks B1 and B3 the identifier 'D' refers to the data element which has been introduced here as 'D'; but inside block B1, for example, 'D' identifies the location 'D * B1'. The reference of 'D * B1' is the same everywhere in the program. Global data elements introduced in relative identifier form need not be introduced again in their associated blocks.

Local data and label elements may always be referred to and introduced by relative identifiers within their own blocks.

Thus the label

L1 * B5)

might be fixed onto an instruction in block B5, L1 would not thereby become a global label. To achieve this, L1 must be incorporated into a global label introduction; such as

block B1,B2(L1,L5,L20),B3,B4(L1),B5(L1);

In order to give a clear distinction between the style of reference, relative identifiers, appropriate to global items and the style, simple identifiers, appropriate to local items, consider the following cases:-

1. If B is a blockname and B is not in use inside this block as a local identifier, then the relative identifier B*B is understood everywhere in the program as a reference to block B.

2. If B is a blockname and B is a global label introduced in this block, then B*B is understood everywhere in the program as a reference to that label.

3. If B is a blockname and B is a local label introduced in this block, then B*B is understood inside block B as a reference to that label, but outside block B as a reference to block B.

2.7 Block Format

Each of the instructions and words in a SAC program must be contained in a block, limited by the control names begin and end. Blocks may not intersect or be contained in each other and every program must be made up of at least one block.

A semicolon must follow the introduction of block identifiers, the introduction of data identifiers, and the control name end. All characters between end and the next semicolon will be ignored by the Assembler provided that the first of these characters is not underline U.

There can be reference to labels in instructions before the instruction they prefix. (Global labels need only have been introduced under the control name block at this time). Similarly, local data identifiers can be used before they are introduced provided that they are declared somewhere in the same block. The introduction of global data identifiers must occur outside the program-blocks, before the block in which they are required.

The contents of a block's local data locations are not overwritten or destroyed on exit from the block since data space is not ~~duplicated.~~ shared between blocks for example, if an identifier A is declared locally in two different blocks, two separate locations are reserved.

2.8 SAC Program Format

Three parts may conveniently be recognised in each SAC program.

(a) The program begins with a 'head', consisting of the definition of the program identifier, followed by the introductions of the block names and global labels, and then by global data introductions.
The global data introductions need only contain those used in the first block, provided the others are introduced between blocks before the block in which they are required.

(b) Then follow the blocks, with global data introductions between them, if necessary.

(c) The program finishes with the control word trigger, followed by a list of the program's entry points expressed as relative label identifiers, or as block identifiers. Elements of the list are separated by commas, and the list is terminated by a semicolon. These entry points must previously have been introduced as global labels (unless they are blocknames).
An example of a trigger list is:
trigger lift* block1, stairs* block1, flats;
SAP causes each SAC program it assembles to be followed by a trigger block of entry points. This contains '4 + n' words, where 'n' is the number of entry points.
An example of the program format is:

```
program example;
block   A, B(L1);
data    P, Q*A, Q*B;
begin   A;
        .
        .
        .
        .
(Label and data introductions, instructions and works)
        .
        .
        .
        .
end;
data x;
begin B;
        .
        .
        .
        .
(Label and data introductions, instructions and words)
        .
        .
        .
        .
end ;

trigger  A, L1 * B ;
```

CHAPTER 3

INSTRUCTIONS AND WHOLEWORDS

3.1 An instruction is a function followed by the address it refers to.

The addresses may be of 5 different types.

(a) INTEGERS (which can be negative), e.g. 51 5 and 00 A/30 - 1 . This type of address should not appear as a reference to a store location, but as a further specification of the function.

(b) RELATIVE, in which the address is regarded as being expressed relative to the address of the first location of the block in which it appears. Such addresses have the form of an integer without a sign followed by a comma.

e.g. 40 15,

which means 'jump to the first half of the 16th location in this block'.

N.B. This method of addressing is not generally used in SAC programs since the position of instructions in locations is usually not known. It is most commonly used when 803 machine code routines are converted into 503 SAC.

(c) IDENTIFIED, e.g. 30 shovel

(d) COMPOUND

(e) DIAMOND-BRACKETED

Addresses of types (d) and (e) are dealt with in Chapter 3.3.

3.2 Wholewords

The content of a location may be expressed in different ways. Any SAC expression of a 39 bit location content is called a 'wholeword'.

Wholewords are divided into the following types:

Integers, Fractions and Floating Point Numbers,
Octal and Alphanumeric Groups,
Order Pairs,
Signed Identifiers.

3.2.1. Integers, Fractions and Floating Point Numbers

These are composed of the digits 0 to 9 and the characters + - . and subscript $_{10}$.

Any whole number between −274 87 7906 944 and +274 877 906 943 inclusive is called an INTEGER and must contain at least one digit and be preceded by a sign.

e.g. +5 −304882

A FRACTION consists of the character . (point) followed by no more than 11 digits: it may be unsigned.

e.g. −.11129811392 .5 +.5

Note that signed fractions such as +0.113 will also be accepted by SAP. Unsigned fractions of the type 0.113 should not be used.

A FLOATING POINT NUMBER is one of the form

$a \times 10^b$

'a' being known as the 'mantissa', and 'b' as the 'exponent'.

The form of the mantissa is: an integer, or a fraction, or an integer followed by an unsigned fraction. It must be such that the number obtained by removing the character 'point' (if present) is an integer as defined here.

The exponent is an integer of no more than 2 digits, and may be without a sign if positive, or even omitted altogether if zero.

A floating point number is written as

$\text{mantissa}_{10}\ \text{exponent}$

and floating point overflow will occur if it is outside the (approximate) range:

$\pm 1.73 \times 10^{-77}$ to $\pm 5.8 \times 10^{76}$

Examples $+3_{10}+5$ (=300,000)

$-7.33192_{10}+60$

$+.892_{10}-27.$

$.5_{10}$

$+0.739_{10}2$

Numbers beginning with the character $_{10}$ will also be accepted, and are understood to have a mantissa of 1.

Zero may expressed as

+0 −0.0 .0 $+0_{10}$ +00000

and so on, but '.000000000000' is erroneous, because it is a fraction with more than 11 digits. '+' is also erroneous because it holds no digit.

3.2.2 Octal Groups

A 39 bit location content may be split into 13 equal parts of 3 bits each. If each of the parts so obtained is expressed as a digit between 0 and 7, and the whole expression is preceded by the digit 8, the result is known as an "octal group".

For example, the number 21 may be written as

80 000 000 000 025

in octal form, since 21 may be expressed as the 39 bit location content

```
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1.1.1
-21 is 87 777 777 777 753
```

3.2.3 Alphanumeric Groups

It is often useful for a SAC program to be able to output headings, titles, error indications, etc. on the typewriter at runtime. Such headings have to be stored in some way as constant data.

The 7-hole character code is punched on tape in an 8-hole form, one of the bits in each character being used for parity checks. Five characters in the seven bit code can be packed side by side into one 39 bit location, (leaving 4 bits spare at the top of the location).

Thus the word 'ERROR' could be packed as:

```
(O)      R        O        R        R        E
. . . .  .11..1.  .1.1111  .11..1.  .11..1.  .1..1.1
```

The SAC expression for the above location is £ERROR

A heading is expressed in Alphanumeric Group form by splitting it up into groups of 5 characters each, each group being preceded by the character £. The number of characters in a heading must be made up to a multiple of 5; this could be done by changing the number of characters in the heading, or by 'filling up' the heading with characters, such as 'run out' (<u>R</u>), which have no effect on print-ups.

Thus the expression for

for heating: £96

as a set of alphanumeric groups is

£for<u>S</u>h£eatin£g:<u>S</u>£9£6<u>RRRR</u>

where <u>S</u> denotes the character 'space', and <u>R</u> has been used to make up the number of characters in the last group to five. Note that the character £ may be used as both a control symbol, defining the beginning of an alphanumeric group of 5 characters, and as one of the characters within an alphanumeric group.

The small section of SAC program which follows will output an alphanumeric word on the output writer. (The word ERROR has been chosen here).

```
          data press;
          30 ERROR
          . . . . . . . . .
again)    20 press
          42 exit
          03 CC127
          17 press/
          74 4096
          51   7
          40 again
          . . . . . . . . .
ERROR) £ERROR
 CC127) +127
```

3.2.4 Reduced Identifiers

An identifier may be packed in reduced form by writing it as a wholeword preceded by the digit 9. i.e. the first six characters of the identifier are packed in one location, as described in section 2.2.1 (RAP). For example,

```
          compar) 9beginning
```

results in the location referred to by the label "compar" holding

```
            b       e       g       i       n       n
. . .    .111..  .11111  1....1  1...11  1.1...  1.1...
```

Note (i) Identifiers of less than six characters do <u>not</u> need to be "made up" to six. Spaces are automatically inserted at the right hand end.

(ii) Spaces within an identifier are packed, not ignored.

3.2.5 Instruction Pairs

An instruction pair is a pair of instructions separated by a B-line digit. If the value of this is 1 (indicated by /) the second instruction is modified by the contents of the location specified by the address in the first instruction; otherwise (indicated by :) the two instructions are wholly separate.

Example. The number 100 may be expressed in SAC as

```
+100
80 000 000 000 144
00 0:00 100
```

3.2.6 Signed Identifier Wholewords

The effect of writing

+IDENTIFIER

in a program location is that the storage address allocated by SAP to IDENTIFIER at Assembly time is placed in that location.

3.3 DIAMOND BRACKET Addresses
COMPOUND Addresses

Whole words may be stored in a SAC program by writing them as part of the program, normally with a label for identification.

e.g. Pi)+3.14159265_{10}

This constant may then be referred to in an instruction such as 30 Pi - the result of which is to place +3.14159265_{10} in the accumulator.

An alternative way of doing this is by the instruction

30 < +3.14159265_{10}>

SAP allocates storage space within the program it is assembling to whole words read in this way. Such wholewords are stored at the ends of the blocks in which they are first read; they are not duplicated in the program if read again, and are available throughout the program.

The instruction

30 < + count >

has the effect of placing the address of the location allocated to 'count' in the accumulator, as opposed to 30 count which puts the contents of the location in the accumulator. This gives the programmer an opportunity to use indirect addressing.

Another example, is the instruction

30 < 9EX 1 >

which results in the placing of the packed identifier, in reduced form, in the accumulator.

A COMPOUND address consists of an identifier address followed by a signed integer.

Example: 30 fork +3

means 'pick up C(fork+3)' and does not mean 'Pick up C(fork) +3.

3.4 Allocation

The process by which SAP determines the reference of an identifier is called 'allocation'.

For example, the data location 'DATA 15' will have been allocated when SAP has determined the address of the particular storage location which it is to identify by this name.

Data identifiers are allocated immediately on being introduced. Label identifiers are allocated at the beginning of the instruction or wholeword following that which they label. This includes block identifiers, which may be considered to label the first instructions of blocks.

3.5 Restrictions on addresses

Identifiers and Blocknames used as labels, if unallocated, may not be used in COMPOUND Addresses.

Diamond Bracketed Addresses may not be used within Diamond Bracketed Addresses.

All invented identifiers used in addresses within a block must have been introduced by the end of that block. It is enough if global labels have been introduced globally by this point.

(See also paragraph 2.8.).

3.5.1 SAP references to unintroduced data items

It is sometimes useful to be able to refer to unintroduced data items by means of compound addresses, as in, for example

```
data    A,B,C(15);
        .
        .
        .
        .
        .
        .
        22  A/
        26  C-1
        .
        .
        .
        .
```

Where the location C-1 falls outside the range of data space which has been reserved by the data introduction, the reference is correctly assembled into store but incorrectly assembled on to paper tape.

The following action will always be assembled correctly, however, since location C-1 now falls inside the allocated data area :

```
data    A,C(15),B;
        .
        .
        .
        .
        .
        .
        22  A /
        26  C-1
        .
        .
        .
        .
        .
        .
```

CHAPTER 4

IDENTIFIERS WHICH ARE NOT CONTROL NAMES

EXAMPLE OF A PROGRAM

This chapter concludes the description of that part of SAC needed to write a complete SAC program. It deals chiefly with those identifiers, not control-names, which SAP recognises as having fixed meanings (standard identifiers). The chapter ends with an example of a complete SAC program.

4.1 Subroutines (standard identifiers LINK, SUBR and EXIT)

SAP automatically allocates a 'link-location' for each block it reads, doing so at the time the block identifier is introduced. Each of these link locations is identified as the location with the global label

LINK

Thus, within their own blocks, these links may be referred to as LINK , and outside as LINK * Blockname , where Blockname identifies the block containing the particular LINK referred to.

Each SAP block may be treated as a subroutine, provided that its exit-instruction is correctly arranged. The standard subroutine entries, to the first instruction of the block, or to a labelled instruction within the block, are then

```
       73 LINK * Blockname : 40 Blockname
    or 73 LINK * Blockname : 40 LABEL * Blockname
```

These may be shortened to

```
       SUBR, Blockname
   and SUBR, LABEL * Blockname
```

respectively: SAP deals with the shortened form as if it had read the instruction pair given above in the normal form.

The standard subroutine exit instruction, to the nth location after that containing the orders causing entry to the subroutine, may be shortened from

```
       00 LINK / 40 n
   to     EXIT, n  or simply  EXIT  if n = 1.
```

Care should be taken, when using these shortened forms of subroutine entry and exit, to make sure that the content of LINK*blockname is not overwritten by a 'recursive use' of the subroutine within itself, or of a subroutine within a block which is itself a subroutine.

4.2. Common Programs (standard identifiers COMP, EXIT CP and LINK CP)

Each SAC program is automatically allocated a link location by SAP, so that SAC programs may call each other up as if they were subroutines. This is done by the order

```
COMP,X
```

where 'X' identifies the program being called. An entry point 'n' may also be specified:

```
COMP,X,n
```

This refers to the nth trigger point declared after trigger in program X. COMP,X and COMP,X,1 are the same order.

The programmer must ensure that program X has a subroutine exit. This is done by exiting from it with the order

```
EXIT CP, n
```

which causes a jump to the nth location after that containing the entry instructions. The common program's link may be referred to as LINK CP ; such references may not be made outside the the common program itself.

It is important that program X be already in the store when SAP is assembling into store the program which calls it.

4.3 Style of Writing SAC Programs
B-digits, dynamic stops, comments

Though the assembled program will be stored two instructions to a location, the instructions are usually written one to a line. Programs written as a series of wholewords will be accepted by SAP, but, from the programmer's point of view, they are more difficult to alter than those in the suggested style.

SAP stores each new order or wholeword in the next available store half-location or location, assuming B-digits to be zero unless given any other indication.

Each order may be followed by one of the B-Digit characters :(= 0) and / (= 1), or be preceded by : . As well as specifying their own value in the instruction pair being assembled, these B-digits also give the position of instructions within computer locations.

Thus 22 wages / indicates that the order 22 wages is to be stored in the next available first-half location, and that it is to be followed there by a bit in the B-digit position.

The two orders

```
30 < +2 > :

24 wages /

30 0
```

would be stored in consecutive locations, as

30 < +2 > : 00 0

24 wages / 30 0

Instructions preceded by colons are understood to be whole words.

e.g. 30 A:

: 20 B

is assembled as 30 A : 00 0

00 0 : 20 B

Note: an instruction-pair written in the form 30 A : 20 B is always assembled into a single location.

Instructions with unspecified addresses are taken to have zero addresses.

e.g. L:71L , L06L and 03 < 77 : >

mean L:71 0L , L06 0L and 03 < 77 0: >

A zero instruction which forms part of a diamond bracketed instruction pair may be omitted.

Thus 30 <77 8191 : 00 0> may be written 30 <77 8191 :>

No SAC program should contain dynamic stops; these are replaced, in the 503 system. by programmed jumps to the Reserved Area Program.

These are performed by using standard identifiers which are written as

STOP - unconditional transfer to RAP

O STOP - transfer if overflow to RAP

N STOP - transfer if negative accumulator to RAP

Z STOP - transfer if zero accumulator to RAP

SAP will ignore comments punched between round brackets, so that

'30 A(A = collating const.)' is read as 30 A.

Comments must not be punched inside any introduction list.

The characters paperthrow P, tab T, backspace B, runout R, space S, and erase E are ignored everywhere except in alphanumeric groups. The character newline L is used as a terminator (any number of these can be used).

N.B. A comment may extend over more than one line.

4.4 Further standard identifiers

Each of the following identifiers represents a subroutine entry to RAP, and each of the subroutines exits to the instruction immediately beneath the entry.

Standard identifier	Function
RAPprint	the reduced identifier in the accumulator is displayed, on the same line.
RAP1print	the reduced identifier in the accumulator is displayed, on a new line. e.g. the instructions 30 < 9EX 1 > RAP1print will cause EX 1 to be displayed.
RAPread	an integer or identifier is read from the typewriter, placed in location 7914 with the terminator in location 7913 and the overflow register is set if an identifier was read. Identifiers are packed in reduced form.
RAPsearch	the reduced identifier in the accumulator specifies a program which is to be searched for in the store. On exit the accumulator holds the address of the program head if the program has been found, but otherwise the content is zero.

4.5 Example of a SAC Program

```
program         square root;

        (this program computes the roots of single length fractions, and precedes the output
        result with a newline and the word ROOT.)


block  root, control (L1);

       (common programs--read, print, and alphanumeric print.)


data   number  *  root;

begin  root;

data   number,  root;


       30     number
       41     L1  *  control
       42     exit
       30     <37 8191/77 8191>
       20     root
L1)    30     number
       56     root          (the formula used is:)
       04     root          (a/w[n] +w[n] - 2w[n+1])
       51     1
       15     root
       41     L1
       07     root
exit)  EXIT, 1
end    root;
begin  control;
       COMP, read
       Z STOP               (return to RAP if end)
       20     number * root
       30     <£ ROOT>
       COMP, alphanumeric output
       SUBR, root
       COMP, print, 2
       00 0/00  11          (print parameter word)
       40     control
L1)    30     <£NEG  R>
       COMP,  alphan
       40     control
end;
trigger control;
```

CHAPTER 5

"REPLACE" STATEMENTS

5.1 This chapter describes how whole sections of SAC may be replaced by a single identifier provided that its meaning is properly defined. One of the simplest examples, is the use of the name EXIT to replace the instruction pair

```
00 LINK/

40  1
```

The name EXIT is called the replacement identifier and the instruction pair referred to is called the replacement text.

Another example is given by the sequence of instructions required to form twice the content of the accumulator plus one:-

```
55  1

04  < +1 >
```

If SOCK is made the replacement-identifier for this replacement text, subsequently, SOCK may be used in place of, and will be replaced by, the above instructions. Thus to place the sum of 2A + 1 and 2B + 1 in a location specified by C i.e. perform

C:= (2A + 1) + (2B + 1) the SAC could be

```
30  A

SOCK

20  C

30  B

SOCK

24  C
```

5.2 Form in which replacements are specified

Specification is in two stages:

(a) Introduction of the identifier

(b) Specification of the replacement text.

The identifier is introduced by the control name replace. The replacement text must follow the introduction unless another means of obtaining the required information is given i.e. a "source" other than the store must be named. SAP recognises certain letters as references to the following sources

	Source
S	the store
R	the second reader
T	the typewriter
Z	none

Thus both stages of the specification must be complete when the replacement-identifier is met during assembly; although stage (b) need not be carried out explicitly if the source is named, so that the program "knows" how to obtain the text.

At assembly time the text is inserted and then SAP proceeds with the next instruction. Identifiers within the text will in general assume meanings current when the replacements are made.

5.3 Replacement introductions - different sources

If no source is given in the statement it is assumed to be S - the store, and the replacement text must follow, enclosed in square brackets. The introduction statement for the previous example should be:

```
replace  SOCK  [ ; 55 1
                      04 < +1 > ] ;
```

A replacement identifier may be associated with a different text by re-introducing it later in the program. For instance its use as a replacement-identifier may be stopped altogether by introducing it as:-

```
replace  SOCK = Z ;
```

which subsequently has the effect of inserting no text at all whenever the identifier appears, thus cancelling any previous text that may have existed.

Several identifiers can be introduced in one statement, e.g.

```
replace  PRINT = R, INPUT = R, Mod 17 = T ;
```

Note that when reference is made to one of the sources, the identifier must be immediately followed by an equals sign and then the appropriate letter.

In the above replacement statement the text of PRINT and INPUT is searched for on the tape placed in the second reader, and they are read in correctly regardless of the order in which they appear on the tape. This provides a convenient method of assembling a program from its constituent subroutines: the functions tape only requiring a single reading.

Whenever SAP encounters a replacement identifier which has been given the source T during assembly, a text is requested, e.g. Mod 17

```
TYPE Mod 17 [
```

is output on the typewriter and SAP waits for a text to be typed, terminated by a closed square bracket. If the operator types the character "=" the data identifier which is the same as the replacement-identifier is taken to be the text (i.e. Mod 17 in this case). Incorrectly typed texts may be cancelled by typing the non-escaping character vertical bar (V), but the text then has to be retyped from the beginning.

Certain identifiers are so placed that they can only be interpreted as replacement-identifiers e.g. identifiers on lines by themselves. These are ignored by SAP if unintroduced when read.

As with data/label identifiers, replacement identifiers may be made local or global by introducing them inside or between blocks. Replacements must not be introduced outside SAC programs.

5.4 Replacement Texts - parameters

Any section of SAC may be used as a text; it can be a single identifier or function, or a complete section including data-, label- and even replacement-introductions. Each text must always be enclosed within square brackets and preceded by the appropriate identifier. It is important that this arrangement is adhered to even when texts are drawn from sources other than the store.

If the same replacement text is used several times in different parts of a program, it is often necessary to be able to change the names of certain data and label identifiers appearing in the text.

The text employs identifiers of three different types:

(A) Label, data and block identifiers

(B) Replacement-identifiers. Replacements may call each other to any depth, although a replacement cannot call upon itself, as this would lead to an infinite recursion - see section 5.5.

(C) Parameter identifiers. These may be "renamed" when the replacement text is inserted into a program. They are referred to as "formal parameters" when in the text, and the renamings and replacements as "actual parameters", using ALGOL terminology (Vol. 2.1.3). Permitted re-namings (actual parameters) are:

(i) an integer, from which the sign may be omitted.

(ii) a constant enclosed within diamond brackets, which is equivalent to the address of a location containing this constant.

(iii) another identifier.

Formal parameters, if any, are enclosed in the same square brackets with their text and need no other introduction. The parameters are listed first and separated from the text by a semicolon. e.g.

```
replace  COPY  [ reader, writer ;

                        again) 06  0

                               71  reader

                               20  work /

                               74  writer

                               40  again ];
```

In this example, 'reader' and 'writer' are the formal parameters.

The separating semicolon must always appear before the replacement-text, even if there are no parameters.

The actual parameters are given when calling the replacement, and are listed after the replacement-identifier enclosed in square brackets, in the order set up by the formal parameter list.

Hence   COPY [2048, 4096]

results in the following instructions being inserted into the program

```
                        again) 06  0

                               71  2048

                               20  work /

                               74  4096

                               40  again
```

Care must be taken when the same replacement text is used more than once in one block, to ensure that labels introduced in the replacement text are made parameters. Also when the replacement is called, the actual parameters for the labels must be different each time the replacement is required within that block. If this is not done the effect produced is that of having two identical labels in one block, which will cause an error.

Thus facilities are provided for the systematic replacement of sections of SAC programming by means of a single identifier, possibly with parameters. The effect of a sequence of such statements can be determined by systematically making the replacements. The expanded text, thus constructed, must obey the rules of syntactically correct SAC. The system, itself, does not check the logical effect of each replacement.

5.5 Identifiers which are given more than one meaning

Identifiers can be of the following types within a SAC program, excluding the program name:

A. Label, data and block identifiers and actual parameters.

B. Replacement identifiers.

C. Formal parameters.

When the same identifier is used as more than one of the above types, its meaning is given in the following order of precedence, Type C, before Type B, before Type A providing each is syntactically allowable. The following examples should clarify this rule:

(i) given the introductions,

```
    data A, B ;

    replace A [ ; B ] ;
```

the instruction 30 A

becomes 30 B

because the replacement identifier A, takes precedence over the data identifier A.

(ii) given the introduction

```
    replace whichone [ ; <+1> ], thisone [ ; <+2> ],

            GET [ whichone; 30 whichone ] ;
```

the instruction GET [thisone]

becomes 30 < +2 >

The identifier whichone is interpreted as a formal parameter, rather than a replacement identifier, so "GET [thisone]" becomes "30 thisone" (not "30 <+1>"). Then thisone is replaced by its text <+2>.

(iii) The same identifier can be used as both a formal and an actual parameter, without producing odd effects.

e.g.

```
    replace PLANT [ SEED, SOIL ;

                    30 SEED

                    24 SOIL ] ;
```

the instruction PLANT [SOIL, POT]

```
becomes      30 SOIL

             24 POT
```

i.e. Although the formal parameter SOIL becomes actual parameter POT this does not affect the actual parameter SOIL.

(iv) A replacements identifier can be quoted in its own text, either explicitly or else as part of the text of one of the nested replacements. Since a replacement cannot call upon itself, the meaning of such an identifier would be taken to be of Type A or Type C. e.g.

given the introductions

```
    data  A ;

    replace  A [ ; 30 A ] ;
```

the instruction A

is taken as the replacement identifier and becomes

```
            30 A
```

where A now represents the data identifier.

(v) A formal parameter only operates within its own replacement-text, excluding any nested replacements inside that text. e.g.

```
    replace  SUB [ ; 53A ] ,  TEST  [ A ; SUB
                                          41 A ] ;
```

```
the instructions  30  A

                  TEST  [ error ]

become            30  A

                  53  A

                  41  error
```

Thus, when determining the effect of a replacement which itself contains a replacement, the text should be expanded one "layer" at a time, applying the appropriate set of parameters to it.

e.g. the introduction

```
    replace  SUB [  B ; 53 B ] ,  TEST  [ A, B, ;

                                        SUB  [ A ]

                                        41  B  ] ;
```

```
with the instructions  30  A

                       TEST  [  <+15>, error ]

    become   30  A

             SUB  [  <+15>  ]

             41 error
```

and then 30 A

53 <+15>

41 error

this rule excludes such a case as

<u>replace</u> HOP [A, B; . . . .] , SKIP [; HOP] ;

with a call SKIP [error, <+1>]

since SKIP has been given actual but not formal parameters.

CHAPTER 6

OPERATING AND PUNCHING INSTRUCTIONS AND ALLOCATION OF STORE SPACE

6.1 Assembling programs

Programs are assembled and run under the assumption that RAP is in its normal position in the Reserved Area: this is important because RAP pointers and subroutines are made use of at assembly-time, and again when the assembled programs are triggered from the typewriter.

All common programs called by the object program must be in the store at assembly time, unless assembly is being made onto paper tape.

Word generator settings

SAP is read in under RAP, by loading its tape in reader 1 and typing:

IN.

The identifier 'SAP' will be output if the tape is read correctly.

To assemble a program, set the word-generator keys in the way described below, load the program in reader 1 and its replacement tapes, if any, in reader 2 and type

SAP.

The word generator keys which are tested by SAP are as follows:

KEY	EFFECT
39 (sign bit)	When the setting is changed during a pause, the assembly continues from the point at which it stopped.
38	When depressed, assembly comes to a pause just inside the beginning of each block, after printing the blockname. The key is tested every time the control name <u>begin</u> is encountered.
37	This key is not normally used when assembling with SAP. If it is depressed, certain information about the identifiers is retained, (this is made use of in SAPTEST for checks).
36	When depressed, at the time of entry to the assembler, key 37 is ignored and the program is assembled on to paper tape. (See section 7 below).
35	When depressed, the program-checksum which is used by RAP, is <u>not</u> formed. Therefore the key should normally only be down if the program alters the content of one of its instruction or labelled (not data) locations at runtime.
34	When depressed, the blocknames are not output during assembly.

Note that a temporary pause will also occur whenever the character & or H̲ (halt code) is read, except when they occur as part of a comment or an alphanumeric group. This allows programs to be punched in parts.

6.2 Messages displayed during assembly Typed replacements

All messages are displayed on the typewriter. The program and block identifiers are output when read, each of them on a new line. The program identifier is followed by the setting of the free store bounds at the time assembly began, and each blockname is followed by the letter x, if the block is being output, letter n in the case of normal assembly and letter c if KEY 37 of the word generator is down.

At each temporary pause (see section 1) the word 'Wait' is displayed on a new line. Assembly ends when the trigger list has been input; END and the new free store bounds are displayed, and RAP then waits for the next message from the typewriter.

The message StFull indicates that the store is full. SAP then continues to input the program tape to find out how much extra store space would be needed to hold the program. This number of locations is output as a 4-digit integer when the end of the trigger list is reached. If there is not even enough room to calculate the extra space, the message NOROOM will be displayed, and RAP will wait for the next typed instruction. The RAP locations FF and LF are not updated until the whole of a program's trigger list has been assembled; the extent of the freestore area is not therefore altered until the program has been completely and, in general, correctly assembled.

SEARCH is output whenever the tape under reader 2 is about to be searched for some replacement text.

The message

TYPE A [

indicates that the text, including formal parameters, of replacement A is required for immediate use. This text should be typed in and terminated with the character] . If a mistake is made while typing, the whole of the text input so far must be cancelled by typing the non-escaping character V̲. The identifier A may itself be given as the replacement text, by typing the character =, (as an example in paragraph 5.3). Typing the character = is therefore equivalent to typing ; A] .

6.3 Error output

If it should happen that SAP detects some error in the SAC which it is inputting, an error indication will be given, and the assembly will continue. In certain circumstances - these are outlined below - it is possible to run such a program.

Error indications are made in the form

ERR x y z

where 'x' is a single letter describing the error-type (see table below), 'y' is a four-digit integer, which gives the number of the line in which the error was discovered, and 'z' is the identifier or instruction producing the error, (a maximum of 13 characters are printed).

'y' is the print up of a line count which is begun afresh, at zero, whenever the control name begin is read. This count is advanced by one whenever a control name, or an instruction, or a wholeword which is not an instruction is read.

x, Displayed letter	Nature of error	Action taken by SAP before it continues assembling	Possibility of running erroneous program
a	Block missing from program when the trigger list is about to be input	If the trigger list is part of a replacement text, outputs NORUN and RAP waits for the next message.	No
		Otherwise comes to a temporary pause (so that the new block may be loaded in Reader 1 and assembly continued: the trigger list must be reinput at the end of the program).	Yes
e	Erroneous character or instruction	Assembles a zero instruction into the object program, and skips to the next newline.	No
	Unintroduced block	Skips to end of block	Yes
i	Identifier given conflicting introductions, (e.g. an identifier introduced twice as a local label).	Ignores new introduction	No
o	Local identifier not introduced by the time end is read	Introduces the identifier as data	Yes (with key 37 down) No (without)
u	Unknown trigger-point given in trigger list	Assembles an entry which causes NOPROG to be output, if used.	Yes, if non-erroneous trigger points used.

In all other cases, the assembly of erroneous programs is concluded with the message NORUN instead of END and the bounds of the free store area are not updated.

6.4 Example of assembler output

(Messages typed by the operator are underlined)

Messages	Comment
IN. SAP	input main assembler under RAP
SAP. >	enter SAP to read
GATHER 2791-5532	the first program. Store Bounds.
apples n	first block - assembled normally.
TYPE count [= TYPE count [; count 1]	typed replacements.
Wait	temporary pause.
truffl c	second block - checked.
END 2860 - 5482	Assembly completed. Bounds.
SAP. >	
SAMPLE 2860 - 5482	
apples n	
truffl n	
ERR e 0064 leave)/	Erroneous instruction in 64th line of block 'truffl', (viz. the instruction begins with /).
ERR u 0093 begin * truffle	
NORUN	End of assembly.
GATHER; > 3.	Enter GATHER via RAP
END	End of run
SAMPLE.	
NOPROG	Sample not in RAP'S list of programs.

6.5 Assembly onto paper tape

The user may instruct SAP to assemble programs, either into the store, so that they are ready for immediate use, or else onto paper tape in a form of binary code. The instruction which causes SAP to assemble the current program onto tape is given by leaving the word generator 36-key depressed as the assembler is triggered; the program is then assembled into the store by blocks, each block being output when it is complete. (Note: Common programs called by the object program do not have to be in store while assembly is being made onto paper tape).

Successive program blocks are assembled into the same store area, so that a program cannot be output to both store and tape in one operation; if the keyboard settings of 36 and 37 are given together, only assembly onto tape will occur.

Output is stopped if any non-ignorable error is detected in the program, or if store space runs out; but the remainder of the program will be input and checked, and the extra amount of store space needed will be calculated, if appropriate.

The output tape is headed with a short leader, a copy of a self-triggering program which will read the assembled program back into store, and then delete itself. (Note: Common programs called by the object program should be in store at this time). This is done by loading the tape in Reader 1, gaining control with RAP, and typing IN. The program will be read and stored in the available freestore area, one of the following messages will be displayed on the typewriter, and RAP will wait for its next instruction:

programname	(indicating that the program has been correctly input).
ERRSUM	(failure of tape checksum)
NOROOM	(not enough store-space for the program)
identi:	
NOPROG	(the common program 'identi' is missing from store)
NOPROG	(this indicates either a software systems error, or that the object tape has been mispunched or misread)

The stored program is triggered by typing

programname; n

where n specifies the required entrypoint.

programname. may be typed in place of

programname; 1.

RAP will sumcheck the program before accepting this entry-instruction; but this sumcheck will not be made if the program's binary tape was produced with the 35-key depressed.

6.6 Form in which assembled programs are stored

Each program is preceded by a 6 location heading and followed by a (4+n) location trigger block, 'n' being the number of trigger points specified in the program's trigger list. The first word is planted in the current first free location and each data item is allocated to the next available location at the opposite end of the free store. Block links are treated as data.

Also, information dictionaries are built up in the area to which SAP allocates data, which extend towards the object program as assembly progresses. Data space is normally allocated so that it overwrites, the dictionaries at runtime, but programs assembled with checks, as described in SAPTEST in Appendix 2, have their data allocated between the dictionary entries, so that information formed by the assembler is still available to the check program at runtime.

Assembled programs are stored in the following form, starting from the low addressed end of the freestore (the current first free location) and progressing towards the high addressed end:

(i) A 5 word program-head (see RAP 2.2.1). The first instruction of the second word in this head is an absolute jump to the first word of the stored program's trigger block.

(ii) A single word, which is used as the program's "common-program-link" (LINKCP).

(iii) Then follows the program proper, stored in exactly the form and layout specified by the programmer; but note -

(a) The first instruction of each block is put in a fresh location; it never shares a location with the last instruction of the previous block.

(b) After each block there may be stored a number of diamond bracket constants, (these are constants which have been used in instructions of the form "FN < . . . . >"). Each diamond bracket constant used in a program is stored at the end of one of the program blocks. If the same diamond bracket constant appears in more than one block, it is stored at the end of the block in which it is first used; unless the constant quotes an unallocated identifier, when it is stored at the end of the block in which this identifier becomes allocated.

(c) STOP, OSTOP, ZSTOP and NSTOP are single jumps and are treated like any other instruction. COMP, EXITCP, SUBR, EXIT are all wholeword instruction pairs

```
COMP, X, n       73 LINKCP of X:  40 } nth entry to X
                                  44 }

EXITCP, n        00 LINKCP/40  n

SUBR, A * B      73 LINK*B : 40  A * B

EXIT, n          00 LINK    /40  n
```

(iv) After the program follows the trigger block, (4+n), words long. This is as follows:

```
        30 RAPsep  :  05  < +31 >

        42      k  :

    Subroutine entry to RAPread

        02      0  :  17  RAPword

      k)42     E1  :  05  RAPword

        42     E2  :  05  RAPword
                   :
                   :
        42     En  :  jump to RAP to output 'NOPROG'
```

where E1, E2.....En are the trigger points.

RAPsep	contains the terminator to the entry instruction typed in by the operator (; or .)
RAPword	holds the word read from the typewriter (in this case the entry point)

Starting from the high addressed end of the freestore and progressing towards the low addressed end:

(i) A single location is allocated for each blockname read which serves as the LINK location for the block. These LINKS are allocated down the store in the order in which their associated blocks appear in the block introduction.

(ii) A single location for each data item introduced again in the order they appear in the introduction lists. No data location reserved in this way is ever associated with any other data name.

6.7 Running SAC Programs

Load any data tapes which may be needed, and type the program identifier terminated with a semicolon. If RAP accepts the entry, it will output a diamond closing bracket > and the operator should then type in the number of the trigger point at which the program is to be entered, terminated with a full stop. Thus, to enter at the nth of the trigger points specified in the program's trigger list, type the integer followed by a full stop.

```
n.
```

Example: a typical entry instruction might be

```
GATHER; > 5.
```

When the run is finished, END will be output, and RAP will wait for its next message.

The typed message

```
GATHER.
          will be understood to mean
GATHER; 1.
```

If the typed message GATHER; causes RAP to display > , an entry is made to the trigger block of the program GATHER. This block examines the terminator and if a semicolon causes re-entry to the integer read subroutine of RAP, to read the number of the trigger point. The trigger block then determines the entry location required.

Entry to programs which are not in the store, or to programs in which irreparable errors were discovered at assembly time, cause RAP to output

NOPROG

instead of > . If the program is present but the trigger block finds the entry is made to an unknown trigger point the same message is typed NOPROG.

6.8 Punching instructions

(i) GENERAL[1]

SAP tapes are punched on the Flexowriter in the Elliott 8-hole code; both tape and print-up should be retained.

Each tape should begin and end with about a foot of runout (blanks).

The program pages should be punched in the order given; about one inch of runout should be added when the end of each page is reached, and ten newlines should be inserted just before punching the underlined word <u>begin</u>.

(ii) TABS

Position the paper so that each newline will begin at about one inch from the edge of the page.

Set tabs at:

12 spaces from margin (to mark the beginning of the ':' - column)

18 spaces from margin (to mark the beginning of the 'address'-column),

and 30 spaces from margin (to mark the beginning of the 'COMMENTS'-column).

(iii) PUNCHING

Ignore all lines which are blank, or which have an X-mark in the 'LINE'-column.

For all other lines, begin by punching everything in the 'INSTRUCTION'-column, tabbing as indicated above.

If the line began with an underlined word, such as <u>program</u>, or if it has the character (in its 'T'-column, then continue punching into the 'COMMENT'-column. Do not otherwise punch this column.

End each line with a newline.

[1]See ELLIOTT 503 SAP PROGRAMMING SHEETS

Control names in which some of the letters are underlined more than once are read as if the letter was underlined once only.

(iv) STANDARD NOTATION

Paper tape code abbreviations such as $\underline{\text{L}}$ should be written as the letter L encircled without underline, thus $\textcircled{\text{L}}$, when giving punching instructions to distinguish them from, for example, 'L underlined'.

o means	the letter O
l	the letter 1
$\textcircled{\text{L}}$	newline
$\textcircled{\text{S}}$	space
$\textcircled{\text{T}}$	tab
$\textcircled{\text{S6}}$	six spaces
$\textcircled{\text{R}}$	runout
$\textcircled{\text{H}}$	halt code

(v) POUND SIGNS

It is important that only the layout characters WRITTEN IN by the programmer be inserted between any £ sign and the newline at the end of the line.

(vi) MAKING CORRECTIONS

Erroneous characters may be overpunched with the erase character.

Lines containing the character £ may only be corrected by erasing the complete line including the £, and then repunching the whole line.

APPENDIX 1

8-CHANNEL PAPER TAPE CODE

00000.000	0	$\underline{R}$	01010.000	32	;	10010.000	64	$\underline{S}$	11000.000	96	?
00010.001	1		01000.001	33	A	10000.001	65	$\underline{F}$	11010.001	97	a
00010.010	2	$\underline{L}$	01000.010	34	B	10000.010	66	$\underline{F}$	11010.010	98	b
00000.011	3	$\underline{P}$	01010.011	35	C	10010.011	67	$\underline{F}$	11000.011	99	c
00010.100	4	$\underline{T}$	01000.100	36	D	10000.100	68	$\underline{F}$	11010.100	100	d
00000.101	5	$\underline{B}$	01010.101	37	E	10010.101	69	$\underline{F}$	11000.101	101	e
00000.110	6		01010.110	38	F	10010.110	70	$\underline{F}$	11000.110	102	f
00010.111	7		01000.111	39	G	10000.111	71	$\underline{F}$	11010.111	103	g
00011.000	8	(	01001.000	40	H	10001.000	72	$\underline{F}$	11011.000	104	h
00001.001	9	)	01011.001	41	I	10011.001	73	$\underline{F}$	11001.001	105	i
00001.010	10	,	01011.010	42	J	10011.010	74	$\underline{F}$	11001.010	106	j
00011.011	11	£	01001.011	43	K	10001.011	75	$\underline{F}$	11011.011	107	k
00001.100	12	:	01011.100	44	L	10011.100	76	$\underline{H}$	11001.100	108	l
00011.101	13	&	01001.101	45	M	10001.101	77	$\underline{F}$	11011.101	109	m
00011.110	14	*	01001.110	46	N	10001.110	78	$\underline{F}$	11011.110	110	n
00001.111	15	/	01011.111	47	O	10011.111	79		11001.111	111	o
00110.000	16	0	01100.000	48	P	10100.000	80		11110.000	112	p
00100.001	17	1	01110.001	49	Q	10110.001	81		11100.001	113	q
00100.010	18	2	01110.010	50	R	10110.010	82		11100.010	114	r
00110.011	19	3	01100.011	51	S	10100.011	83		11110.011	115	s
00100.100	20	4	01110.100	52	T	10110.100	84		11100.100	116	t
00110.101	21	5	01100.101	53	U	10100.101	85		11110.101	117	u
00110.110	22	6	01100.110	54	V	10100.110	86		11110.110	118	v
00100.111	23	7	01110.111	55	W	10110.111	87		11100.111	119	w
00101.000	24	8	01111.000	56	X	10111.000	88	[	11101.000	120	x
00111.001	25	9	01101.001	57	Y	10101.001	89	]	11111.001	121	y
00111.010	26	10	01101.010	58	Z	10101.010	90	$_{10}$	11111.010	122	z
00101.011	27	11	01111.011	59		10111.011	91	<	11101.011	123	
00111.100	28	-	01101.100	60		10101.100	92	>	11111.100	124	
00101.101	29	+	01111.101	61		10111.101	93	↑	11101.101	125	
00101.110	30	−	01111.110	62	$\underline{V}$	10111.110	94	~	11101.110	126	$\underline{U}$
00111.111	31	.	01101.111	63		10101.111	95	%	11111.111	127	$\underline{E}$

Abbreviations

$\underline{B}$	Backspace	$\underline{F}$	Friden Programmatic Codes
$\underline{E}$	Erase	$\underline{H}$	Stop Code (Halt)
$\underline{L}$	New line	$\underline{P}$	Paper Throw
$\underline{R}$	Run Out	$\underline{U}$	Underline
$\underline{S}$	Space	$\underline{V}$	Vertical Bar
$\underline{T}$	Tabulate		

APPENDIX 2

SAPTEST

FUNCTION

SAPTEST is a version of the SAC translator which includes the following facilities in addition to all the facilities of SAP. These extra facilities are provided to assist in the testing of SAC programs.

(1) A trace of labelled instructions (with or without output).

(2) Checks on the intermediate values of data variables on reaching any specified labelled instructions.

(3) A 'dump' of intermediate results, together with the current values, of the accumulator and modifier, auxiliary and overflow registers, in sumchecked binary, at any specified labelled instructions.

(4) Entry to the program being tested at any labelled instruction setting the values of data variables and the registers from a previously obtained 'dump' i.e. the 'load' facility.

Note: in (2), (3) and (4) the check, dump, or load is carried out before the labelled instruction is obeyed.

METHOD OF USE

Entry

A test run is controlled by the entry point or points made to SAPTEST. A prepared 'control tape' is then used to supply the information needed to perform the desired tests.

Entry is made to SAPTEST in the normal RAP manner; by typing the program name, (SAPTEST); followed by the entry point, (M). Also certain word generator settings are required.

The eight entry points to SAPTEST are tabulated below together with their functions:-

	M	Functions
Program translation	1.	Entry to the SAC translator within SAPTEST. This has exactly the same effect as an entry to SAP.
Specification of type of checking required.	2.	Test run with the output of checks as specified on a control tape.
	3.	Extra data for a check run (load next control tape).
	4.	Test run with a trace of all the labels, (output being dependent upon the settings of key 37 and key 1 of the word generator) preserving the current last ten labels traced.
	5.	Test run incorporating the facilities of both 2 and 4.

	M	Functions
Operations at run time	6.	Output the last ten labels traced.
	7.	Transfer control back to the program being tested, to continue with a normal run.
Begin test run	8.	Entry point for start or continuation of run (see Operating Instructions).

Entry 3 may be used at any time after an entry 2, 4 or 5 has been made to SAPTEST; all information obtained from previous control tapes will have been preserved, e.g. if after a test run it is decided that insufficient checks have been made, then further checks may be specified on another control tape and entry 3 used to input these extra specifications (see Operating Instructions).

Entry 6 may be used at any time during or after a test run (if necessary, manually interrupting the run). SAPTEST will subsequently display on the output writer, the last ten labels traced.

The word generator settings tested by SAPTEST are as follows:

KEY	EFFECT
39 (sign bit)	same as for SAP (see paragraph 6.1)
38	" " " " " " "
37	When depressed the current block is assembled with checks, (i.e. certain information is retained which can later be used by SAPTEST) provided that the key was down when reading the program head and that key 36 is not depressed. This key is tested immediately after key 38 so that check-setting may be altered at the beginning of each block. If any of the checking facilities of SAPTEST are to be used on a block, key 37 must be depressed as described during assembly. During a trace, the output may be permitted or suppressed at any time by setting or clearing this key.
36	same as for SAP (see paragraph 6.1)
35	" " " " " " "
34	" " " " " " "
1	If depressed the only labels output are those to which control is transferred non-sequentially during the run. If clear, all labels traced are output.

Control Tape

The following information should be specified on a control tape.

(i) The name of the program to be tested, together with the entry point and the output device to be used.

(ii) A list of blocks to be made non-checkable areas, which have not been so specified during translation. (see word generator settings).

(iii) The point at which the test run is to end, if other than the normal point. (Only one point may be specified).

(iv) Specification of any 'dumps' or 'loads' required.

(v) Specification of any checks required.

N.B. The entry point of SAPTEST used will over-ride any information specified on a control tape, e.g. if checks are specified on a control tape and entry 4 to SAPTEST is made, no checks will be output.

Punching Instructions

The control tape is made up of these five parts which are punched in the following form.
N.B. Italics are used to indicate that the appropriate identifiers should be inserted for these items

(i) *program name* ; N; J.

where N and J are unsigned integers.

N is the entry point of the program being tested.

J indicates the output device to be used as follows:

J = 1 for output on punch 1

J = 2 for output on punch 2

J = 3 for output on the output writer.

(ii)
```
blockname
blockname
    .
    .
    .
blockname
```

(iii) ↑ *label* ∗ *blockname*

(iv) *label* * *blockname* ; *control word* ; *data* ;

```
                                         data ;
                                           .
                                           .
                                           .
                                         data ;
```

where the control word is either dump or load. Any number of specifications of this type may occur on a control tape and the last item of data, for a dump or load list, is terminated by '.' (full stop) instead of ';' (semicolon).

(v) *label* * *blockname* ; *data* ; I.

```
                          data ; I.
                            .
                            .
                            .
                          data ; I.
```

end

Any number of specifications of this type may occur on a control tape where end terminates the list of data for a check and I specifies the format in which the value of the data variable will be output:

I = 1 fractions

I = 2 integers

I = 3 octal

I = 4 pseudo instructions

I = 5 floating-point

If fraction, integer or floating-point format is required a common program PTSPRINT must be in the store at run time (see below).

N.B. The value of an identifier which has been replaced may be output, but the identifier in the replacement text and not the original replacement identifier must be specified on the control tape.

e.g. If it is wished to output the value of the replacement identifier "number" and at label "LX" the identifier "number" has been replaced, by "zahl", part (v) of the control tape should read:

LX * *blockname* ; zahl ; I.

Notes

1. A control tape must be terminated by the character %.

2. For SAPTEST entries 2, 4 and 5 the control tape will consist of (i) followed directly by (ii) and then (iii), (iv), (v) in any order. For entry 3, (i) and (ii) should not appear on the control tape as they have previously been specified.

3. *data* may be either a data variable, a part or the whole of an array or a label. A data array may be punched as

   data name (n : m)

   or in cases of local variables *data name* * *blockname* (n : m)

   where $0 \leqslant n \leqslant m \leqslant$ array max.

4. If I is to have the same value as was last specified on the control tape, then *data* may be followed by a full stop instead of a semicolon. I is then omitted from the tape.

5. The following characters are ignored by SAPTEST:

   space, tab, erase and redundant new lines.

Restrictions

1. The minimum amount of information which must appear on a control tape is part (i)

   i.e. For a trace run the control tape need only be:-

   program name ; N;J.

   %

   The maximum amount of information on a control tape is limited by the amount of free store avaliable. An approximate indication of the store required for data on a control tape is given by:-

   $F = 20 + 2i + b$

   where i is the number of identifiers on the control tape (<u>not</u> the number of different identifiers).

   b is the number of blocks in the program being tested.

   F is the number of locations required by SAPTEST to store the information from a control tape.

2. If a label is specified as the end of a test run, then other specifications referring to that label will be ignored.

3. SAPTEST will not accept on a control tape

   label ±n as an identifier

   or *data name* +n as a data variable (where n is an integer)

4. If both a dump and a check are required at the same label, then the specification of the dump must precede the specification of the check.

Common Program PTSPRINT

For its use within SAPTEST the common program PTSPRINT must conform to the following specifications.

Entry will be made via the first trigger point with two parameter words P1, P2 following the entry instruction. Only the F1 bits of P1 and the N2 bits of P2 need have specific meanings for the purposes of SAPTEST, but if PTSPRINT assumes meanings for the remaining bits of the parameter words, it should be noted that in their use within SAPTEST they will always be clear.

The F1 bits of P1 will specify the format to be used:-

F1 = 00 for integers

F1 = 04 for fractions

F1 = 40 for floating point

The N2 bits of P2 will specify the output device to be used:-

N2 = 0 for output on Punch 1

N2 = 2048 for output on Punch 2

N2 = 4096 for output on the output writer.

N.B. The formats corresponding to I = 1, 2 and 5 need not be as above (i.e., fractions, integers, floating-point). Any desirable form of output may be obtained by using a suitable version of PTSPRINT.
A 503 library program PTSPRINT is available which conforms to the above specifications (see 2.2.3.20).

Operating Instructions

(1) Input RAP using the initial instructions (see 3.1.4.1.).

(2) Input SAPTEST by typing IN.

(3) Translate all necessary common programs.

(4) Translate the program with checks on all necessary blocks. (See the word generator settings).

(5) Load the control tape in reader 1 and any dumped tape required in reader 2.

(6) Enter SAPTEST at the required entry point.

program name ; N is output on the specified device.

DATA WAIT is output on the output writer.

N.B. If the control tape is one of extra data for checks then only DATA WAIT is output.

(7) Load any data tapes required by the program being tested and then enter SAPTEST at entry point 8.

N.B. All SAC programs concerned in a test run must be translated using SAPTEST even if a standard SAP is in the store at the time of the test run.

Messages displayed during a Test run

(i)	DATA WAIT	is displayed after the control tape has been read. Control is returned to RAP.
(ii)	dump AT *label* ∗ *block* *data* . . . *data*	is displayed during execution of a dump.
(iii)	load AT *label* ∗ *block* *data* . . . *data*	is displayed during execution of a load.
(iv)	END OF CHECK RUN *program* WAIT	is displayed when control is returned to RAP by the program being tested.
(v)	CHECKED TO *label* ∗ *block* *program* END	is displayed when a label is reached which has been specified as the end of a check run.

After the messages (i), (iv) or (v) have been displayed control is returned to RAP to await further messages from the input typewriter.

Error Indications

Message	Cause of error		Subsequent effect
	During input of control tape	During run	
TAPERR	incorrect character on the control tape or sum check failure when a dumped tape is loaded.	none	RAP wait.
DICERR	error connected with SAP dictionaries e.g. a blockname has been read where a label was expected.	none	RAP wait
ERROR	unknown identifier on control tape	dump at a label in an uncheckable area.	RAP wait.
OVERLP	insufficient space in the store to hold the data from the control tape	insufficient space in the store to hold the current last ten labels.	RAP wait
NOPROG	the program specified on the control tape is not in the store.	none	RAP wait
NOCHECK *label* ∗ *block*	none	a label in a non-checkable area specified for a check.	test run continues.

FORM OF OUTPUT

Both trace and checks are output on the device specified on the control tape.

Trace

Each label is output on a new line together with its associated blockname unless it is the same label as was last traced; (i.e. a simple loop is being executed) in this case the blockname will be followed by an unsigned integer, enclosed in round brackets, indicating the number of times the label has been consecutively traced.

e.g.
```
begin  *block1

data   *block1  ( 64)

invert *block1

COMP

output*block2

title  *block2

data   *block2  ( 64)

end    *block2
```

COMP indicates that control has been transferred to a point outside the bounds of the program being tested.

N.B. With the limited output of trace, (that controlled by key 1 of the word generator) labels are output on a new line whether they are repeated or not.

Checks

The label at which the check is occurring is output on a new line followed by the data name required and the current value of the data variable in the required format, each on a new line. If an array is output each element is output as an individual item of data being punched:-

e.g. *data name* + n
actual value

```
label1*block3

INCREA*block3

+ 73

DATE

0000104307712

TREND + 1
```

```
00 57:10 193

TREND + 2

00 65:10 199

TREND + 3

00 83:10 221

PROFIT

40  0:03 8162
```

If fraction, integer, and/or floating-point format is called for and the common program PTSPRINT is not in the store, then the current value of the data variables concerned will be output in octal format and the corresponding data names will be followed by the character &.

Dump

The binary tape produced during the execution of a dump will be output on Punch 1. The first non-zero character on the tape will have a binary value of 80. This will be followed first by the values of the specified data variables, then by a zero word and finally by the current values of the accumulator and modifier, auxiliary and overflow registers followed by a checksum.

Example of a control tape (with explanatory notes)

```
EXAMPLE; 3;2.                          program name; entry; output device

block1                                 uncheckable block
block4                                 uncheckable block
↑label2*block3                         test run terminating point
label1*block2;dump;COST*block2;        output in binary the contents of: local data COST;
                   PROFIT;                                        global data PROFIT;
                   COUNT;                                         global data COUNT;
                   AREA(1:25)                                     global array AREA.

label1*block3;INCREASE*block3;2        output: local data INCREASE as an integer;
              DATE;3                           global data DATE in octal;
              TREND(1:3);4                     global array TREND in pseudo orders;
              PROFIT                           global data PROFIT in pseudo orders;
              A*block3(1:2);4                  local array  A  in pseudo orders.
end
%                                      terminating character of control tape.
```

CONFIGURATION

The basic 503 computer.

SPEED

SAPTEST reduces the speed of the program being tested by a factor of at least 60, depending upon how much output is made by SAPTEST in the form of traces and checks.

TAPES

A sum checked binary tape is provided suitable for input by RAP.

EXAMPLE

The following is an example of a program, a suitable control tape and data tape and the resulting output during a test run using SAPTEST.

program

```
program  EXPONENTIAL;              (Computes  e ↑ x)
block    read,
         master (start),
         print;

data     EXP, DELTA, SUM;

begin    read;                     (input  x  and maximum)
  .
  .                                (permissible error, storing)
  .
  .                                (x  in EXP and error/2 in DELTA)
  .
end      read;

begin    master;
data     COUNT,LAST;
start)   30      <+1.0₁₀>
         20      SUM
         20      COUNT
         30      EXP
repeat)  20      LAST
         60      SUM
         20      SUM
         30      LAST
         10      COUNT
         60      <+1.0₁₀>
         10      COUNT
         63      EXP
         64      COUNT
         61      DELTA
         41      end
         60      DELTA
         40      repeat
end)     30      SUM
         40      print
end      master;

begin    print;
  .
  .
  .
  .
end      print;

trigger  read;
```

control tape

```
        EXPONENTIAL; 1;2.
        read
        print
        repeat*master;COUNT*master;5.
                     LAST*master.

        end
        end*master;  EXP.
                     DELTA.
                     SUM.

        end
        %
```

data

```
        +.04
        +.00005
```

Operator action	*Displayed message*	*Output on Punch 2*
SAPTEST ; 5		EXPONE: 1
	DATA WAIT	
SAPTEST ; 8		start *master
		repeat*master
		COUNT*master
		$.10000000_{10}+01$
		LAST *master
		$.00000000_{10}+00$
		repeat *master
		COUNT*master
		$.20000000_{10}+01$
		LAST *master
		$.40000000_{10}-01$
		end *master
		EXP
		$.40000000_{10}-01$
		DELTA
		$.25000000_{10}-04$
		SUM
		$.10408000_{10}+01$
	END OF CHECK RUN EXPONE	
	WAIT	
SAPTEST ; 6.		start *master
		repeat *master(2)
		end *master
	END	

APPENDIX 3
ASSEMBLY-TIME EDITING FACILITY

All versions of the assembler, from issue 3 onwards, include a facility which allows the program text to be edited while it is being assembled into main store. The editing is achieved by means of an 'edit tape' which contains instructions of the form specified in the descriptions of EDITALL or EDIT8.

To make use of this facility the following sequence of operations must be observed:

1. Input the SAP assembler and EDIT8 (in either order).
2. Load the program tape in reader 1.
3. Load the edit tape in reader 2.
4. Trigger SAP at its second entry point:
   type SAP;2.

503 TECHNICAL MANUAL
VOLUME 2: PROGRAMMING INFORMATION
PART 1: PROGRAMMING SYSTEMS
SECTION 3: ALGOL MK. 1

January 1965
(*Issue 3*)

ELLIOTT
SCIENTIFIC COMPUTING DIVISION
Elliott Brothers (London) Limited
Elstree Way, Borehamwood, Hertfordshire, England

First Printed	August 1963
Reprinted	October 1963
Issue 2	March 1964
Issue 3	January 1965

CONTENTS LIST

2. Input and Output

PREFACE

This specification describes how to write programs in Elliott ALGOL MARK I for running on the Elliott 503 computer.

The name MARK I does not imply a new compiler, but is used merely to distinguish it from the new 503 ALGOL MARK II compiler developed for use on a configuration which includes some form of backing store: either core storage or magnetic tapes.

It is assumed that the reader is already familiar with ALGOL 60. For those requiring an introduction to the language the following publications are recommended:

H. BOTTENBRUCH
Structure and Use of ALGOL 60
Journal of the ACM, April 1962

E. W. DIJKSTRA
A Primer of ALGOL 60 *Programming*
Academic Press 1962

D. D. McCRACKEN
A Guide to ALGOL Programming
Wiley 1962

R. W. WOOLDRIDGE and J. F. RACTLIFFE
Introduction to ALGOL 60 *Programming*
E.U.P. 1963

Appendix III of the latter appears by kind permission of the authors and publishers.

A basic knowledge of the Reserved Area Program is also assumed as this is the most fundamental program on the 503. See section 2.2.1.

CHAPTER 1

SYSTEM SPECIFICATION

1. ELLIOTT 503 ALGOL REPRESENTATION

The reference language of ALGOL requires a set of 89 printable characters; 78 of these characters are directly available in the 8-channel Elliott telecode and a further 4 can be reproduced by punching a combination of two characters. This leaves only 7 characters which are not available.

The following section gives the necessary changes which must be made in an ALGOL program before it can be presented for punching in Elliott 8-channel telecode.

Note. All bold type used in the text indicates that the words should be underlined when used in an ALGOL program. This does not apply to paragraph headings (e.g. 3.6.1. **wait**).
Also the word 'display' is used to mean 'print out on the output writer'.

1.1. Basic Symbols

503 representation	ALGOL *equivalent*
and	$\wedge$
or	$\vee$
not	$\neg$
£	‘
?	’
div	$\div$
*	$\times$

To give compatibility with 803 ALGOL, the following alternative representations are acceptable, but not necessary, in 503 ALGOL.

representation	ALGOL *equivalent*
less	$<$
gr	$>$
lesseq	$\leqslant$
greq	$\geqslant$
noteq	$\neq$
equiv	$\equiv$

The synonyms **boolean** for **Boolean** and **goto** for **go to** are also acceptable in 503 ALGOL, but ** for $\uparrow$ and @ for $_{10}$ (subscript ten) are not acceptable.

Note. The symbol $\supset$ is not allowed (see 2.14. of this chapter).

1.1.1. The Characters halt (76) and %

The characters ⊕ (halt) and % are reserved in Elliott ALGOL for special purposes connected with tape editing. They should not appear in an ALGOL statement or on a data tape, or in a comment, unless the corresponding special effect is required. In a string, % is treated in the same way as any other character, but ⊕ should not be used (see paragraphs 1.6 and chapter 2 paragraph 1.3).

1.2. Punching Instructions

1. An ALGOL program tape should be punched on an 8-channel Flexowriter. The punched tape should then be printed up and this print up checked against the original program script.

2. ALGOL programs should be written either on a pre-printed form (the Elliott ALGOL Program Sheet – see end of 2.1.3) or on lined paper with some vertical lines added. The heavy vertical lines represent tabulation and indicate where each line of print should commence. The tab. stops should be inserted every six character positions from the left hand margin, which, in turn, should be set approximately $\frac{3}{4}''$ in from the left hand margin of the paper. This permits eight stops to be inserted when using a standard paper roll.

3. Punch exactly what is necessary to produce a print-out like the written program, i.e. all blank lines, spaces, etc. Consecutive underlined words should be separated by a space. In general the exact number of spaces, new line characters and tabulate characters is not critical, but punching the correct number improves the general appearance of the print-up. However, between the characters £ and ?, the text must be punched exactly as written.

4. Care must be taken to avoid confusion between the figure '1' and the letter '*l*' and also between the figure '0' and the letter 'o'. These must be punched correctly and punch operators should familiarise themselves with the difference in print of these characters.

5. There should be a run out of blank tape at the beginning and end of every tape punched. Blanks can also be run out at any time.

6. A wrong character may be cancelled by overpunching with 'Erase'. This may occur anywhere.

7. Every semicolon should be followed by three or more blanks to simplify future editing of the tape. (The omission of these blanks is not an error.)

8. A halt code ⊕ should be punched at the end of every program tape, or at the end of every physical piece of program tape if the program is punched in parts, and also at the end of every data tape.

9. If it is impossible to punch one line of manuscript on one line of paper, a new line may be commenced between any two consecutive words, numbers or symbols. A single word must not be split.

10. To produce an underlined word, punch the underline character before each letter of the word. Do not punch the underline character after the last letter of the word. To produce one of the following symbols, punch the pair of characters shown opposite it:

Symbol	*Characters Punched*
$\neq$	Vertical bar =
$\equiv$	Underline =
$\leqslant$	Underline <
$\geqslant$	Underline >

Since 'underline' and 'vertical bar' do not move the carriage, this produces the correct print-up.

11. If **goto** is punched as two words then the space must also be underlined.

1.3. Notes to the Programmer

1. ALGOL programs may be written on a pre-printed form (503 ALGOL PROGRAM sheet) or on lined paper (on which a series of vertical tabulation lines have been drawn).
2. Consecutive words (identifiers) or consecutive basic words must be separated by at least one space and manuscripts must be written with such words clearly separated.
3. The programmer must clearly differentiate between the figure '1' and the letter '*l*' and also between the figure '0' and the letter 'o'. The use of the continental 1 and 7 and a script '*l*' is strongly recommended.

1.4. The Program

Every ALGOL program written for the 503 must be preceded by a title and followed by a semicolon.

A title consists of any string of characters not including a semicolon or a halt code, and is terminated by a semicolon. This title is displayed when the program is run.

The title should be used to give enough information to identify the program and the programmer uniquely. It may include name and department of the programmer, cost code of the work being done, and the date on which the program is presented. Each installation is encouraged to establish its own standard practice for the writing of titles. Since the title is not part of the program, a comment may not come immediately after the semicolon which ends the title.

1.5. The Use of Elliott ALGOL Program Sheets

Elliott ALGOL program sheets have been designed to enable the programmer to indicate to the punch operator the exact layout of his program. For this purpose, one and only one character should be written in each cell; a cell which does not contain a character is treated as a space if it occurs in the middle of a line; after the last occupied cell of a line a change to a new line is punched.

The cells are grouped in units of six by means of a firmer line. This is an aid to punch operators in counting the number of spaces required on a deeply indented line. It is recommended that indentation should be a multiple of six; this allows room for the tabulation of statements just to the right of a **begin.**

For identification purposes, the name of the program should be written at the head of each sheet; if the sheet contains a recognisable subsection of the program, a subtitle may be used. These names at the head of the sheets are not reproduced on the punched document. Therefore the first sheet of the program must contain a copy of the title written as part of the text of the program as well as at the head of the sheet.

1.6. Correction of ALGOL Programs

Any errors in an ALGOL program, whether of a syntactic nature or in the actual formulation of the problem, must be corrected in the source language. There is no means of making corrections to the compiled program in the store of the computer.

To insert a statement or group of statements into a program after a given statement, S say, punch ⊕ (76) on the tape immediately after the semicolon that terminates S. On reading this the computer comes to a systems wait. Then translate the additional statements (the last one having ⊕ punched after the terminating semicolon), and finally continue translating the original program.

To cause the compiler to skip a statement of the program, punch % immediately after the semicolon that terminates the preceding statement. On reading % the computer ignores all subsequent input until the next semicolon.

The halt code facility also permits a program to be punched in sections on separate tapes, each section having ⊕ at the end.

According to the punching instructions, three or more blanks should be left after every semicolon of the program. If this has been done, it is an easy matter to insert ⊕ or % by a hand punch wherever necessary.

Note. In most cases it is easier and more convenient to modify the ALGOL text before translation by using the 503 Library Program EDITALL (see 2.2.3.21.).

2. RESTRICTIONS AND PROGRAMMING NOTES

This section describes a number of restrictions imposed on the full generality of ALGOL, and mentions restrictions which are part of ALGOL and whose effects are frequently overlooked.

RESTRICTIONS

2.1. The Declaration of Labels

Any labels used to label a statement in the compound tail of a block must be declared at the head of that block. The method of declaring a label is to include it in the **switch** list of a **switch** declaration:

switch ss := label1, label2, ...;

The **switch** identifier is obligatory, even if it is not used in a statement of the program. Any identifier different from the other identifiers of the program may be used for a **switch**; but the use of a sequence of the letter 's' is recommended as standard practice, except where this would cause a clash with other identifiers.

Care must be exercised in declaring labels in the blocks to which they are local. It is inadmissible to declare labels of an inner block in an embracing block. The following is a scheme of correct label declarations:

```
begin ....
      switch s := l1, l2, l3;
        .
        .
        .
        .
      l1: .............;
        .
        .
        .
        .
      l2; begin switch ss := l3, l4;
                 .........
             l3: ........;
                 .....
             begin comment compound statement;
                   .......
               l4: .......;
              end compound statement
           end block l2;
           .
           .
           .
           .
      l3: .............;
           .
           .
end;
```

A label may not prefix a statement in a procedure body unless it is declared in the same procedure body. This may involve turning the statement which forms the procedure body into a block, by attaching a **switch** declaration, and, if necessary, a **begin** and **end.**

2.1.1. Unsigned Integers as Labels

Unsigned integers may not be used as labels. Where they occur in a program, they should be turned into identifiers. The recommended practice is to precede each number by a sequence of one or more '*l*'s.

For example, '23:' might become '*l*23:' or '*ll*23:'.

2.1.2. Switches

The elements of a **switch** list may only be labels. These labels must be prefixed to statements of the block in the head of which the **switch** declaration occurs. The occurrence of a label in a **switch** list serves as a declaration of that label and therefore no label can occur more than once in the **switch** declarations of any one block.

If, in a published program, these conditions are not satisfied, the **switch** declaration must be replaced by a portion of program which achieves the desired effect. It should seldom be necessary to do this and no general rules are given.

If a **go to** statement uses a **switch** designator, and the subscript of the **switch** is non-positive, or greater than the number of labels in the **switch** list, the computer displays *Subscr Oflo* and no further statements of the program are obeyed. If this is expected to occur, the **go to** statement should be turned into a suitable conditional statement. For example,

go to ss [i]

where the **switch** list of ss has three elements, would become:

if i $>$ 0 **and** i $<$ 4 **then go to** ss [i] ;

this has the same effect as that specified in the Revised ALGOL report.

2.2. Type of Arithmetic Expressions

If the type of an arithmetic expression depends upon the evaluation of an expression or upon the type or value of an actual parameter then it is taken to be **real**, e.g. the result of exponentiation is always **real**, even when both arguments are integral. Provided that the result is within the range $-536\ 870\ 912$ to $+536\ 870\ 911$, no accuracy is lost as a result of this.

If the result of exponentiation appears on either side of **div** (the integer division sign), the translator will not accept the program and will display *Error no. 31* to indicate an inadmissible operation. To correct this, the standard function 'entier' should be used.

e.g. n $\div$ 2$\uparrow$ m
should be written as
n **div** entier(2$\uparrow$ m)

2.3. go to statements

go to statements occurring inside a procedure body may not have as their destination a statement which is outside the procedure body, thereby causing an exit of the procedure; however, a **go to** statement may lead to a label which is an actual parameter of the procedure. Label parameters may not be called by **value**.

There are two methods of adapting a program to evade this restriction:

(i) adapt the procedure declaration and procedure calls to specify the label as a parameter
(ii) an 'error exit' may be turned into an 'error procedure' which performs the necessary printing, and then enters the standard procedure 'stop', which stops the program.

2.4. for statements

The implementation of the **for** list element is not exactly as described in section 4.6.4.2. of the Revised ALGOL Report, in that the address of the controlled variable is computed only once at the start of the statement, and is not re-computed each time round the loop.

2.5. own arrays

The subscript bounds of an **own array** must be written as integers. There are two courses of action which can be taken for the few programs which violate this rule:

(i) the widest feasible bounds should be chosen and the integer values inserted in the declaration
(ii) the array should be made into a non-own **array** of a block surrounding the block in which the original **own array** declaration was made.

2.6. Specification of Parameters

All formal parameters of a procedure must be specified in the head of the procedure.

2.7. Recursive Procedures

A recursive procedure may not have any **real, integer** or **Boolean** parameters called by name. However, a procedure may be entered recursively during the evaluation of any of its parameters, whether called by name or called by **value**.

2.8. switch parameters

A formal parameter of a procedure may not be specified as a switch; consequently, a switch identifier may not appear as the actual parameter of a procedure, though a switch element may do so.

The easiest method of evading this restriction is to use an output name parameter to specify the subscript of the switch.

2.9. Procedures as Parameters of Procedures

A formal parameter of a procedure may not be specified as a procedure. Consequently no, procedure identifier, alone without parameters, may appear as an actual parameter, unless it is a parameterless procedure which stands as an expression.

This restriction may be evaded by use of name parameters.

2.10. Type Procedures

A call of a type procedure can only occur in an expression: it cannot stand alone as a procedure statement.

2.11. Sequence of Declarations

Any number of declarations may occur in any order in a block head, but other declarations may not occur after a procedure declaration. All procedure declarations of a block must be grouped together at the end of the block head.

A procedure body may contain activations of any other procedure declared previously in the same block head. Any set of procedure declarations may be put into sequence in such a way that this rule is satisfied, except in the case of mutual recursion.

2.12. Length of Identifiers

Identifiers are distinguished by their first six characters; any subsequent characters are ignored. If an identifier coincides over its first six characters with any other concurrently used identifier, its name must be changed. No general recommendation can be made on a method of systematic alteration; it is advisable to restore uniqueness by making a reasonable abbreviation of the word. For example, the identifiers 'condensation' and 'condenser' appearing in the same program might be abbreviated to 'condsn' and 'condsr'.

2.13. Reserved Identifiers

The identifiers 'checkr', 'checki', 'checkb', 'checkB', 'checks', 'location', 'elliott' and 'precompile' are reserved for special purposes and may not be used in any other way (see chapter 3).

2.14. The Operator $\supset$

The Boolean operator $\supset$ (denoting implication) is not allowed in Elliott ALGOL. The Boolean expression

$$A \supset B$$

may always be replaced by the equivalent expression

B or not A

PROGRAMMING NOTES

2.15. Range and Accuracy of Numbers

In all cases where an operation is defined as having a result of type **integer**, this result must be in the range

$$-274\ 877\ 906\ 944 \text{ to } +274\ 877\ 906\ 943$$
$$-2^{38} \text{ to } 2^{38} - 1$$

If an integer exceeds these limits, the program stops and the message *Int oflo* is displayed.

In an arithmetic expression a constant is treated as a positive number with an associated sign. Thus in the example:

```
        begin integer x;
                    x := − 274877906944 ;
                    :
                    :
```

integer overflow results.

In all cases where an operation is defined as having a result of type **real**, this result must be in the range

$$-5.79 \times 10^{76} \text{ to } +5.79 \times 10^{76} \text{ approx.}$$
$$\text{i.e.} \quad -2^{255} \quad \text{to} \quad 2^{255} - 2^{226}$$

If a real number exceeds this limit during the running of a program the computer stops and ERRINT 1 is displayed. During translation, a real constant that is too large is replaced by the largest possible real constant ($\simeq 5.79 \times 10^{76}$).

Where any operation (including reading and printing) is defined as having a result of type **real**, this result may be inaccurate by up to two parts in 10^9. It is not recommended to expect an accuracy greater than eight significant decimal digits in testing convergence of a numerical process.

If a number with exponent greater than 12 is read, a slightly greater inaccuracy may be expected. In particular, if two such numbers a and b are read, and a is greater than b by an amount which is too small to be represented, then the machine representation may be such that b is greater than a.

2.16. The Standard function abs(E)

In accordance with the ALGOL 60 Report, this function operates on an argument of type **real**. If I and J are variables of type **integer** the statement I: = abs(J) involves a double type conversion, and if J lies outside the range −536 870 912 to +536 870 911 rounding occurs. This effect may be avoided by substituting for 'abs(J)'

if J **less** 0 **then** −J **else** J

2.17. The Standard trigonometrical procedures

The arguments of sin(E), cos(E) and tan(E) are expressed in radians. If, within the limits of the representation E is an odd multiple of $\pi/2$, then tan(E) is assigned the value of the greatest representable positive number ($\simeq 5.79 \times 10^{76}$).

The inverse functions satisfy

$$-\pi/2 \leqslant \arctan(E) \leqslant \pi/2$$
$$-\pi/2 \leqslant \arcsin(E) \leqslant \pi/2$$
$$0 \leqslant \arccos(E) \leqslant \pi$$

In extreme cases the results of these operations are indistinguishable from 0, $\pm\ \pi/2$ and π. As a result of the finite accuracy of the representation both limiting values can occur in each case.

2.18. Sequence of Operations

The order in which operations are performed within an arithmetic expression is undefined except insofar as it is determined by the rules of precedence:

(1) exponentiation
(2) multiplication and division
(3) addition and subtraction

Thus in

a : = b + c + d

the order of evaluation of b,c and d is undefined so that if one of them is a type procedure which affects the value of one of the others (a 'sneaky' procedure), the value of a will be undefined. If it is important that b,c and d are evaluated in that order, they should be converted into expressions by the use of brackets:

a : = (b) + (c) + (d)

Similarly, if it is important that the two additions are performed in the order shown, the expression should be written:

a : = (b + c) + d

If a, b, and c are **real**, and are of widely differing magnitudes, then (a+b)+c may not be equal to a+(b+c).

2.19. Boolean Expressions

In the evaluation of a **Boolean** expression, every term is evaluated. Thus, in the expression

A : = B **or** C **or** D

even if B is found to be true, and thus the value of the expression determined, C and D are nevertheless still evaluated. Hence any side effects from C and D always occur.

Remarks analogous to those of 2.17 apply to the order of operations within a **Boolean** expression.

2.20. Correspondence between formal and actual parameters

In most cases if the formal and actual parameters of a procedure are of different type, conversion of the actual parameter automatically takes place. However, in certain cases the correspondence between formal and actual parameters must be exact. These are as follows:

2.20.1. Array Parameters

If a formal parameter of a procedure is specified to be an array, the corresponding actual parameter must in all cases be of the same type and have the same number of dimensions.

2.20.2. Parameters called by name

If a formal parameter of a procedure is called by name and within the procedure body has a value assigned to it, then the corresponding actual parameter must be a variable of the same type. It must not be a constant or an expression.

No error indication is given during translation or running. Any attempt to assign a value to a constant or an expression at run time normally results in the loss of the assigned value.

CHAPTER 2

INPUT AND OUTPUT

1. INTRODUCTION

ALGOL 60 provides no method of programming a computer to input the data for a calculation or to print out the results. In view of this, each implementor is obliged to adopt a system of his own.

1.1. print and read statements

The system chosen for Elliott ALGOL is based on the introduction of two new types of statement, the print statement and the read statement. The syntax of these statements is very simple, since they consist only of the underlined words **print** or **read**, followed by a list of operands, the items of the list being separated by commas, e.g.

read x, y, b[j];

print $x \uparrow 2$, x*y, cos(x/q) − b[j];

The effect of a **read** statement is to cause one or more numbers to be read from some input device, and assigned (in the sequence read) to the variables specified in the list. The effect of a **print** statement is to cause the values of the arithmetic expressions occurring in the list to be output (in the order given) on an output device of the computer.

1.2. Structure of read and print lists

The elements of a **read** or **print** list may be arithmetic variables or procedures. A **print** list may also contain arithmetic expressions and strings. These elements are scanned in the sequence in which they are written, with the following effect:

item	*effect in a read list*	*effect in a print list*
arithmetic variable	A number is read from the current input device and its value is assigned to the named variable	The value of the variable is printed on the current output device.
number or arithmetic expression	NOT PERMITTED	As for an arithmetic variable.
non-type procedure	The procedure body is executed	The procedure body is executed.
arithmetic procedure (inside the body of the procedure)	As for an arithmetic variable	This is a recursive call of the procedure; the value assigned to the procedure by this call is printed on the current output device.
arithmetic procedure (not inside its own body)	NOT PERMITTED	The procedure body is executed and the value of the procedure is printed on the current output device.
Boolean variable or procedure	NOT PERMITTED	NOT PERMITTED.
string	NOT PERMITTED	The string is printed on the current output device.

1.3. Input Data Tape

Numbers to be read by the computer should conform to the ALGOL definition of number, and each number must be followed by some character other than a digit, a decimal point or subscript ten. This terminal character may be followed by any sequence of characters excluding the digits 0 to 9 and the characters . $_{10}$ £ + − . The sequence may be of any length; its only function is to separate the numbers on the data tape, and it is otherwise completely ignored. Note that spaces may not normally be used in the middle of numbers, since they are classified as terminating characters (but see 2.5.3. of this chapter).

Blank tape is ignored under all circumstances. The character 'halt' in any position on the input tape causes the computer to wait.

The character £ is not permitted on a data tape, except in connection with the 'instring' facility (see 2.6. of this chapter).

The example data tape	*is read as a succession of numbers:*
item 176329	176329
197 ft 7 ins	197
	7
g = 1.276_{10}–11	1.276_{10}–11
7.392_{10}–4	7.392_{10}–4
710623 1.49700	710623
	1.49700

1.4. Presumed Settings

The **print** and **read** statements of the simple type explained above are executed under the control of the presumed settings; methods of changing the settings and obtaining varied and more sophisticated effects of format control is described in 2.3. of this chapter.

All numbers read under control of the presumed settings are read from tape reader number one, and all numbers output on tape punch number one.

Numbers printed under presumed settings appear on separate lines with up to eight digits. The values of integer expressions less than 100 000 000 are printed with leading zeros replaced by spaces and, in the case of a negative number, the sign floated, i.e. the sign is moved along so that it immediately precedes the most significant digit.

E.g. − 00000252 is printed as
−252

Larger integers are printed in a form similar to that of large real numbers.

The values of real expressions, if less in absolute magnitude than 100 000 000, are printed as decimal numbers containing eight digits and preceded if necessary by a minus sign. Larger numbers are printed with an exponent part and only four significant digits.

Examples of printing under presumed settings:

Integers	*Real Numbers*
9	−9.0000000
−127	127.30614
239610	−23961.000
−14306971	.00000000
$-2.75_{10}+11$	$-1.364_{10}+22$

1.5. Output of Text

In order to output headings and other messages, the characters which comprise the message should be enclosed within the string quotes £ and ?, and included as an item of the list of a **print** statement; these items being output in the sequence in which they appear.

Examples:

```
print £ height            weight            speed?;
print F, £ ft?, i, £ ins ?;
```

To output lines, spaces, blanks, or the symbols £ and ? , an inner string should be used which consists of certain special interpreted characters enclosed by two pairs of string quotes (£ and ?). These characters may appear inside single string quotes amongst text which is itself enclosed in string quotes. The meanings of the special characters are given in the following table:

Character	*Interpretation*
l	new line
s	space
r	blank (i.e. runout)
q	£(i.e. quote)
u	?(i.e. unquote)
t	tabulate
h	stop

All the above characters will be accepted on upper case as well as lower case.

If any of these characters is followed by an unsigned integer, the effect is the same as writing the character the specified number of times.

Spaces, tabulation symbols and changes to a new line may also occur within a single pair of string quotes, and they are reproduced in the same way as other characters; care must be taken to ensure that only those characters required are quoted.

Examples:

```
print ££l4?chapter?, n, ££l2s5??;
print ££l?height £s12? weight £s12? speed?;
```

Text is output on the same device as a number in the same list. Under presumed settings, therefore, output will be on punch one. Text is not preceded by a change to a new line, and therefore is printed on the same line as the last number. A change to a new line may be specified as part of the text.

2. SETTING PROCEDURES

The presumed settings are adequate for inexperienced programmers, for program testing and low-volume output. In many applications, however, the programmer may wish to use all available input and output devices, to tabulate his results, and to exercise control of output format. For these purposes a number of procedures (setting procedures) are provided.

A setting procedure statement normally occurs in the list of a **print** or **read** statement, separated in the normal way by commas from the other operands. All numbers and text in a list subsequent to a setting procedure statement will come under the control of the settings specified. Any number occurring previously in the list or in the lists of other **print** statements, is totally unaffected. Thus operations in a list previous to any setting procedure statement are always executed under the control of the presumed settings (see 1.4. of this chapter).

A setting procedure statement may also occur as a separate statement of the program. In this case, once it is obeyed it will be effective until it is cancelled by a contradictory setting procedure statement. Such a cancellation may occur locally inside a **print** statement, or globally as a separate statement of the program.

Where a parameter of a setting procedure is of type **integer**, the actual parameter may be an expression, allowing the format of results to be determined dynamically.

2.1. Device Setting Procedures

In order to use input and output devices other than the first reader and punch, it is only necessary to mention the name of the device required in the list, prior to the operands concerned.

The names of the input and output devices are:

reader(1)	punch(1)
reader(2)	punch(2)
reader(3)	punch(3) (i.e. the typewriter)

The **print** statement

```
print x, punch(2), sum, £ = sum ?, punch(3), £feed next tape?;
```

causes the value of x to be printed on the first punch (the presumed setting), the value of sum, and the message '= sum' to be printed on punch 2, and the message 'feed next tape' to be displayed for the attention of the operator.

2.2. Prefix Setting Procedures

The procedure 'sameline' should be used to print numbers on one line; this will suppress output of the prefix (new line) normally output before each number.

The statement

```
print t, £tons?, sameline, c, £cwt?, l, £lbs?;
```

causes the value of t to be printed on a new line (the presumed setting), followed by the rest of the numbers and text on the same line. e.g.

```
17tons          12cwt          19lbs
```

If all the numbers to be printed on one line are to be preceded by the same character or characters, the procedure 'prefix(£..... ?) should be used. This causes the text displayed between £ and ? to be printed (in the place of new line) before every number. If the procedure 'prefix' is used, the procedure 'sameline' is unnecessary. In fact, 'sameline' will cancel the effect of 'prefix' if it occurs after it in the list, and vice versa.

For example, to print five numbers on a line separated by a comma and two spaces, write the statement

```
print a, prefix(£,£s2??), b, c, d, e;
```

2.3. Format Setting Procedures

The presumed format settings are satisfactory for numbers up to a hundred million in absolute size, with an accuracy of eight significant figures. If the range of numbers to be printed would be greater or less than this, or if a greater or lesser accuracy is required, it may be desirable to change the number of digits printed. To specify the number of digits in an integer expression, write 'digits(n)' in the list just before the items to be printed with n digits, where $1 \leqslant n \leqslant 12$.

For example, if i, j, and k are integer variables, the statement

```
print i, digits(4), j, k;
```

could produce the output:

```
    -213
-1024
  362
```

The first integer is printed under control of the presumed setting (8 digits), while the other two have up to four digits, and therefore take less room in the column.

The procedure 'digits' has no effect on the printing of the values of real numbers or real variables. To achieve the same effect with real numbers, use the procedure 'freepoint(n)', where $1 \leqslant n \leqslant 9$; this has no effect on the printing of integers. For example, the statement

print x, freepoint(4), y, z;

could produce the output:

```
1.9632450
176.1
.0032
```

2.4. Scaled and Aligned Formats

When a column of numbers is to be printed, it is often desirable to position the numbers in such a way that the decimal points of all the numbers in the column are aligned one beneath the other. Elliott 503 ALGOL provides two methods of doing this. The scaled format always places the decimal point after the first digit, and uses an exponent part to indicate the scale of the number. The aligned format allows the programmer to specify the number of digits to be printed before and after the point.

To obtain the scaled format with n digits, write 'scaled(n)' as a setting procedure, where $1 \leqslant n \leqslant 9$; to obtain aligned format with m digits before the point and n digits after, write 'aligned(m,n)' as a setting procedure, $(m + n \leqslant 15)$; to revert to the freepoint format with n digits, write 'freepoint(n)' as a setting procedure. Note that more than one setting procedure may appear in a **print** statement; if the setting procedures contradict each other, the later ones cancel the effect of the earlier. Thus the statement

print x, prefix(£, ?), scaled(4), x, aligned(4,3), x, freepoint(4), x

would cause the following sample output if it were obeyed repeatedly:

```
-1.2345678,     -1.235₁₀+00,    -1.235,     -1.235
 123.45678,      1.235₁₀+02,   123.457,     123.5
 .00123457,      1.235₁₀-03,     0.001,     .0012
```

2.5. Additional Format Setting Procedures

2.5.1. Grouping of digits

The special format setting procedure 'grouping(n)' should be used to obtain a spacing of digits in printed numbers. The digits of numbers output under control of this setting procedure are grouped in sets of n, the sets being separated by a single space. The origin of the grouping is the decimal point for real numbers, and the least significant digit for integers.

This procedure may be used to improve the legibility of numbers which consist of a long series of digits. It is not recommended for use with the freepoint format. Tapes produced under the control of 'grouping(n)' can be reinput only under the control of special(4) (see 2.5.3. of this chapter).

The following examples illustrate the effect of the procedure statement 'grouping(3)' on the output of integers, and of real numbers in the aligned and scaled formats.

```
 1 076 253      12 073.193 6     1.207 319₁₀+04
   -92 765     -   120.965 0     1.209 650₁₀+02
   -   142     -24 306.701 4     2.430 670₁₀+04
12 630 441          12.000 0     1.200 000₁₀+01
```

2.5.2. Leading Zeros

Integers are normally printed with leading zeros replaced by spaces and the sign floated. The setting procedure statement 'leadzero', allows the programmer to specify any character to replace these leading zeros. It also suppresses the floating of the sign.
E.g.

normal print format	*setting procedure statement*	*resulting print format*
−9	leadzero (£0?)	−00000009
176	leadzero (£*?)	*****176
−9	leadzero (£?)	−9
−12345	leadzero (£?)	−12345
−12345	leadzero (£ ?)	− 12345

Note that if the actual parameter is a null string, the significant digits and the sign if negative, are the only characters printed. Also the characters £ ? blank and erase may not appear within the string (i.e. an inner string is not allowed).

2.5.3. Specialities

The design of output documents sometimes requires certain special effects which may be obtained by the use of the procedure 'special(n)', ($1 \leqslant n \leqslant 4$).

special(1) causes the space normally output before positive numbers to be suppressed. It should not be used when printing negative numbers.

special(2) causes a plus sign to replace the space normally output before a positive number. It should be used if the numbers are separated by spaces and are to be reinput under special(4).

special(3) causes any sign output not to be floated.

special(4) affects reading only, and causes spaces, normally treated as terminating characters, to be ignored. This facility enables tapes with grouped numbers to be reinput.

special(1) cancels special(2) and special(3), and vice versa. special(2) and special(3) can coexist.

2.5.4. Miscellaneous

buffer(n,£α?) is a Boolean function which takes the value true if, and only if, the current character in the buffer for input device n is the character α, α must be a single character but not £, ?, blank, or erase. The use of an inner string is not allowed in this case. The tape does not move in the reader.

advance(n) has the effect of moving the tape in input device n on by one character, overwriting the character currently held in the buffer for that device.

2.5.5. Errors

If an attempt is made to print a number that is too large for the style of printing called for, e.g. to print 289.2 in the aligned(2,4) format, or to print 12347 in the digits(4) format, then 'alarm printing' occurs in such a way that the layout of the printed page is undisturbed. Provided that the total number, n, of characters called for is greater than six, the number is printed in scaled($n-6$) format; if $n \leqslant 6$, the letter H and a halt code are printed, preceded by ($n-1$) spaces.

An attempt to print a real number that is not in standard floating point form causes the message *Print error* to be displayed.

2.5.6. Parameters out of range

If the parameters of a format procedure are out of the permitted range, a return to presumed settings occurs as follows:

Procedure called	*Permitted range of parameters*	*Presumed setting occurring*
digits (E)	$1 \leqslant E \leqslant 12$	digits (8)
freepoint (E)	$1 \leqslant E \leqslant 9$	freepoint (8)
scaled (E)	$1 \leqslant E \leqslant 9$	freepoint (8)
aligned (E,F)	$1 \leqslant E+F \leqslant 15$	freepoint (8)
grouping (E)[1]	$1 \leqslant E$	prefix(££*l*??), leadzero (£ ?) Cancels grouping, special(1), special(2) and special(3)
special (E)	$1 \leqslant E \leqslant 4$	as for grouping but special(4) is also cancelled.
reader (E)	$1 \leqslant E \leqslant 3$	reader (1)
punch (E)	$1 \leqslant E \leqslant 3$	punch (1)
advance (E)	$1 \leqslant E \leqslant 2$	advances on reader one
buffer (E, £ ?)	$1 \leqslant E \leqslant 4$	tests buffer of reader one.

[1] If, in grouping (E), E is greater than 63 the style of printing is undefined.

2.6. Input and output of strings

Strings may be read in as data, and printed out again, by using the procedures 'instring(A,m)' and 'outstring(A,m)', where A is a single-dimensional integer array, and m is an integer variable.

The effect of 'instring' is to search on the current input device for a £; if a numeric character is encountered during this search, the message *Read error* is displayed. The string that follows the £ is input and stored in A[m], A[m+1],..... When the ? which brackets with the original £ is encountered, input stops, and the variable m is set to the index of the next available element of A. Thus, on repeated execution of the procedure, m is automatically set in such a way that the strings cannot overwrite each other.

The effect of 'outstring' is to output a string which has been previously input by the instring facility. The integer variable m indicates the index of the first of the elements of A which contain the string, and the procedure will set m to the index of the first element containing the next string. Thus, on repeated execution of the procedure, successive strings stored in the array A are output in succession. Inner strings are interpreted in the same way as that specified for strings occurring in **print** statements.

The array A should be sufficiently large to contain all the strings read. A string of n characters will occupy $(n + 4) \div 5$ locations.

In any call of these procedures the second actual parameter must be an integer variable (which may be subscripted). It may not be a constant.

2.7. Procedures in read and print lists

The body of a procedure occurring in a **read** or **print** list may contain **read** and **print** lists and setting procedures. On entry to each **read** or **print** list the current read format or print format settings respectively are preserved. They are restored on exit. Thus the input and output of numbers during the execution of a procedure can be controlled externally by putting the procedure call into an appropriate **read** or **print** list.

2.8. Note: input by 'elliott' orders (see chapter 3.4.2.) or other means not involving the ALGOL statements **read**, 'instring', or 'advance' will leave the value of the ALGOL internal buffer for the appropriate device unchanged. To avoid trouble after any such input an 'advance' statement should be obeyed before the next **read** or 'instring' statement for the appropriate device.

CHAPTER 3

ADDITIONAL STANDARD PROCEDURES

1. INTRODUCTION

In addition to the standard functions described in the ALGOL 60 report, certain other functions and procedures are available to users of Elliott ALGOL without explicit declaration.

2. ADDITIONAL STANDARD FUNCTIONS

arccos(E)	:	result in range 0 to π
arcsin(E)	:	result in range $-\pi/2$ to $+\pi/2$
tan(E)	:	the tangent of the value of E

3. CHECKING FUNCTIONS

The standard checking functions provide the programmer with a means of optional printout of intermediate results during the course of a calculation.

There are three such functions:

checkr	for real argument
checki	for integer argument
checkB	for Boolean argument (the spelling checkb is also accepted)

and they are written in the program as functions with a single parameter, e.g.

```
A := checkr(B+2*C)+1.5;            (1)
```

If the B-digit on the keyboard is not depressed during translation, the checking functions are ignored—that is to say, the statement (1) is treated exactly as if it were

```
A := (B + 2 * C) + 1.5;
```

If the B-digit is depressed during translation, then extra orders are compiled so that, if the B-digit is depressed also during the running of the program, the value of the argument is output on the current device. Thus (1) would have the effect of assigning to A the value of B+2*C+1.5, and of printing out the value of B+2*C.

In the case of Boolean checking, the output is one of the words *true* or *false*. In all cases check output is preceded by a change to a new line and an asterisk.

The checking functions may be nested, so that it is possible to write

```
A := B + checkr(M[checki(i + 3)]);
```

A := B + M[i+3];

in which case the value of (i + 3) would be output before the value of M[i + 3].

In addition to the three checking functions, there is a checking procedure

checks

whose parameter is a string. It can conveniently be used to provide a form of 'trace' of the progress of a calculation, or to identify the output of the checking functions. A typical call of this procedure would be

```
checks(£ stage 1 complete?).
```

Groups of checked output can be spaced by using the procedure call checks (£?), which outputs a newline character and an asterisk only.

4. OPTIMIZATION PROCEDURES

In certain applications of ALGOL it is considered highly important to produce an efficient program. While the ALGOL translator in general produces machine code programs of reasonable efficiency, there are certain occasions where ALGOL prevents efficient use of the computer. For this reason, some facilities, similar to those available in machine code, have been introduced in Elliott ALGOL.

4.1. Addressing Facilities

The addressing of subscripted variables is one of the less efficient aspects of ALGOL. Elliott 503 ALGOL therefore provides facilities which enable the programmer to write his own address calculation routines in a notation very close to that of ALGOL, without necessitating a knowledge of the machine code of the 503.

In order to do this effectively, it is necessary to know the method of storing arrays in the computer. A variation of one in the last (rightmost) subscript of an array variable corresponds to a variation of one in the address of the variable. A variation of one in the other subscripts corresponds to a variation in address equal to the product of the ranges of all subsequent subscripts. Thus, in the case of a matrix, storage may be said to be by rows.

E.g. Given the array A[1:8, 1:9, 1:10]
and two elements A[3,4,5] and A[3,5,5]
the difference in the addresses of the locations in which they are held = 10.

In order to compute the address of a subscripted element of an array, use may be made of the following standard functions, which have integer values:

address(A)	the address of the first element of the array A;
lowbound(A,I)	the declared lower subscript bound for the Ith subscript position from the left in the array A;
range(A,I)	the number of values in the range of the Ith subscript position from the left in the array A;
size(A)	the number of elements in the array A.

The first element of A is the subscripted variable all of whose subscripts take values equal to their lower bounds. The range of a subscript is computed by subtracting the lower bound from the upper bound and adding one. The size of an array is equal to the product of the ranges of its subscripts.

If, in a call of lowbound or range, the value of I lies outside the range

$$1 \leqslant I \leqslant \text{number of subscripts of A},$$

the message *Subscript oflo* is displayed.

The following table gives the formulae which may be used to calculate the addresses of subscripted variables:

variable	*address*
A [I]	I - lowbound(A,1) + address(A)
A [I,J]	(I - lowbound(A,1))*range(A,2) + J - lowbound(A,2) + address(A)
A [I,J,K]	((I - lowbound(A,1))*range(A,2) + J - lowbound(A,2))*range(A,3) + K - lowbound(A,3) + address(A)

When the required address has been calculated, a method must be available for accessing the variable corresponding to the address. For this purpose the subscripted variable 'location [I]' has been provided. This variable, which is assumed to be of type **real**, must have as its subscript an identifier of an unsubscripted integer variable. Subject to these restrictions, it may appear like any other variable in an arithmetic expression or in the left hand part of an assignment statement.

The facility for explicit address calculation will be most profitable in the following cases:

(i) the same subscripted variable occurs several times in close proximity

(ii) array variables with the same subscript bounds appear with the same subscripts several times in close proximity

(iii) one or more variables used as subscripts are controlled by nested **for** clauses, and none of the relevant subscripts is altered by assignment within the loop. In this case the necessary increment to the address may be computed outside the loop and there is no need to resort to a multiplication inside the loop.

4.2. Machine Code

There are certain operations which may be performed very efficiently when expressed in the machine code of the 503, which cannot be performed efficiently when they are specified in ALGOL. This is because the ALGOL translator must cater for all possibilities, but the programmer writing in machine code, need only consider his special case. Therefore a facility is included for writing machine code instructions in an Elliott ALGOL program.

Machine code is particularly effective in cases where the following operations occur in inner loops:

(1) multiplication or division of integers known to be positive

(2) multiplication and division of positive integers by a power of 2.

The use of machine code imposes on the programmer the responsibility for testing and clearing overflow where necessary, and of ensuring that the program itself is not interfered with by a B-lined instruction. It is strongly recommended that all programs should be written exclusively in ALGOL during the testing stages, and that when some of the operations are replaced by machine code, all the tests should be run again to ensure that no coding error has occurred.

4.2.1. elliott

One word of machine code may be introduced into an Elliott ALGOL program by use of the standard procedure 'elliott' which has seven parameters. The first and second parameters represent the first and second function digits of the first instruction in the word, and the actual parameters must be unsigned integers in the range 0 to 7. The third parameter represents the address part of the first instruction, and the actual parameter must be either an unsigned integer, a label, a variable, or an array identifier. The fourth parameter represents the B-digit and the actual parameter must be either 0 or 1 (unsigned). The fifth, sixth, and seventh parameters represent the second instruction of the word and are subject to the same rules as the first, second, and third parameters.

The procedure statement 'elliott' causes a whole word of machine code to be inserted in the object program. Those parameters which are unsigned integers are replaced by the corresponding binary number in the appropriate function, B-digit or address positions. An identifier is replaced by an address at time of translation.

The following table gives the meanings of the address:

identifier:	*are replaced by:*
simple variable (or formal parameter called by value)	The address of the location which stores the value of the variable.
array (or formal parameter specified as an array)	The address of a location which contains the address of the location which stores the first element of the array.
label	The address of a location which contains a jump to the instructions corresponding to the statement which has been prefixed by the label. This jump may be in either half of the location but if it is in the second half, the first half will contain a do nothing instruction. Only the functions 40, 41, 42 and 43 may be used.

Note. it is not permitted to refer to a formal parameter called by name (other than an array) or to a procedure or switch.

4.2.2. example

To multiply a by 2^b, setting the result to zero if integer overflow occurs (a and b are of type **integer** and b is non-negative).

machine code	*elliott procedure*
55) 30 0,3 : 00 0	elliott (3,0,a,0,0,0,0) ;
56) 00 1,3 / 55 0	elliott (0,0,b,1,5,5,0) ;
57) 47 58, : 20 0,3	elliott (7,3,4,1,4,7,2) ;
58) 40 59, : 26 0,3	elliott (2,0,a,0,4,0,dog) ;
	elliott (0,0,0,0,2,6,a) ;
	dog: rest of program

Notes

(1) If a label is attached to a call of elliott a jump to that label will always cause transfer to the first instruction of the instruction pair.

(2) Location 4 is always available for temporary work space in a series of elliott procedures, since it is similarly used by many of the dynamic routines. Information left in it should be assumed to be lost if anything other than a label separates two calls of elliott.

(3) The integer overflow indicator is tested at each entry to the dynamic routines. Since these entries are frequent it is advisable to clear it, if there is any possibility of the overflow indicator being set by a series of calls of elliott. This can conveniently be done by placing

elliott (7,3,4,1,4,3,1) ;

at the end of the series.

(4) When location 4 is used in this way it is permissible to construct a jump to the second half of a location, as is done in the above example.

5. SPACE ALLOCATION

The ALGOL system on the 503 is controlled by the Reserved Area Program (RAP).

The exact position in the store taken up by the system on input depends on what other programs are already in the store.

When a program has been translated and is ready to be run the available store is divided into three areas:

(i) at the low numbered end is the compiler, occupying approximately 4,800 locations.

(ii) at the high numbered end is the translated program together with all the routines necessary for its running—e.g. **print, read,** standard procedures, etc. The space occupied by these routines is approximately 1,600 locations.

(iii) the space in between is called the *Free Store.*

The Free Store area is used by the program mainly for arrays, other declared variables having been allocated space at translation time.

Arrays are allocated space dynamically at the time of entry to the block in which they are declared—from the high numbered end of the free store backwards. If more space is needed than is available in this area, the compiler is overwritten.

5.1. The FREE STORE message

The message 'Free store = *n to m*' is displayed at the end of translation; *n* and *m* are the addresses of the beginning and end of the Free Store area. Thus the space available for arrays—without overwriting the compiler—is approximately $m - n$ locations. If the compiler is overwritten, the space available is approximately $m - n + 4800$ locations.

5.2. storemax

The standard integer procedure 'storemax' takes as its value the number of locations of free store currently available for use by an array of maximum size (overwriting the compiler). The declaration for this array would be:

begin array A [1 : storemax];

If any other arrays are declared in the *same* block head or in an inner block, or as workspace of a procedure called in the block, the combined size of these arrays must be subtracted from storemax; otherwise space overflow may result.

The activation of recursive procedures or procedures in which a local array is declared, is also likely to cause space overflow in a block whose arrays already occupy most of the available space.

6. CONTROL PROCEDURES

Elliott ALGOL provides a number of standard procedures to assist in the operating of a program; most of them do not have any effect on the actual calculations. Where the use of these procedures requires the co-operation of the computer operator, use should be made of the Operation Sheet (see Chapter **5.2.2.**).

6.1. wait

This procedure causes the operation of the program to be held up until a signal is given by the operator to continue. It should be used in cases where some action is required from the operator (e.g. changing paper or magnetic tapes). The operator must be informed what action is required by an entry under *Special Instructions* on the Operation Sheet.

6.2. restart

This procedure causes a transfer of control back to the beginning of the program. It should be used whenever it is expected that the same program will be run several times with different sets of data.

If the 'restart' instruction is omitted from a program, and it is required to run the program several times, the operator may be asked to make a manual restart at the end of the program (OS 16).

6.3. stop

This procedure causes an immediate end of the program, thereby freeing the computer for the next program to be run. However, it is still possible to make a manual restart.

6.4. dump

To avoid the necessity of translating an ALGOL program every time it is run, it is possible to obtain a version of the program, together with all current workspace (i.e. declared variables and arrays), on a binary tape. This binary tape can be read into the computer by RAP, and it is not necessary to have the ALGOL translator in the store at the time.

To obtain the binary tape, write in the program the procedure statement 'dump'. When this statement is obeyed, the binary dumped program tape will be produced. When this tape is reinput, it will start operating at the statement *immediately following the dump*, and will operate from then on, just as if no dumping had occurred. Note that if the 'dump' statement in an already dumped program is obeyed again, it will produce yet another dumped program tape.

Note also that a program that has run to its end cannot be re-entered via RAP at the statement following the dump. To restart calculation again at this point, an appropriate **go to** statement (possibly preceded by a wait) should be written at the end of the program. The program may, however, be started right from the beginning by a manual restart.

It is essential that the operator be informed of the presence of a dump instruction in a program (OS 13). It is possible to continue running a program after the dumped version has been produced (OS 14). If the dump instruction has been omitted, a manual dump may be obtained (OS 11); this is possible only before the running of the program has begun.

The programmer can control the dumping externally by using the keyboard of the computer. For example, when the following set of statements is obeyed, a dump will take place if and only if the least significant keyboard digit is depressed.

```
                    i := 1;
                    elliott (7,0,0,0,0,3,i) ;
                    elliott (4,2,no,0,0,0,0);
                    dump ;
                no:
```

6.5. precompile

To avoid translating the whole of an ALGOL program at one time, it is possible to split the program into two parts, A and B, and to translate A on one occasion and B on another. This is particularly useful if the first part of the program is a set of procedure declarations, common to many different programs.

To use this facility, write the procedure statement 'precompile;' at the end of part A, and write a title (terminated by a semicolon) at the beginning of part B.

Part A is translated in the usual way. When the statement 'precompile;' is encountered, translation stops and a precompiled version of A is produced on paper tape. Then, if this tape is input before B is translated, the effect is as if the complete program (A + B) had been translated

When a precompiled package is used, the ALGOL system must be in the same position in the store as it was when the package was prepared.

'precompile' can be used only when translating in the normal (load and go) mode. Therefore, part A must not be so long that it cannot be translated load and go—although the combined program(A + B) may be. Only one precompiled segment at a time can be added to a final segment to make a complete program. However, it is possible to add to a precompiled segment A1 the statements contained in another segment A2 (which ends with 'precompile;') by inputting A1 and then translating A2.

In the case where part A is a general piece of program that is used with several second parts B—for example, a package of procedure declarations—particular care must be taken that each B contains the necessary **end**'s to terminate all the blocks and compound statements of A that are not terminated in A itself.

'precompile;' is a statement, and therefore, if part B of the program begins with a declaration, this must be preceded by a **begin**, thereby starting a new, inner block.

An example of the form which the two parts A and B may take is given below. D stands for a declaration, and S for a statement.

```
    Name A;                          ⎫
    begin D1; D2; D3;                ⎪
    S1; S2;                          ⎬ A
    begin D4;                        ⎪
    S3; precompile;                  ⎭
    Name B;                          ⎫
    begin D5;                        ⎪
    S4; S5; S6                       ⎪
    end block of B                   ⎬ B
    end second block declared in A   ⎪
    end first block declared in A;   ⎭
```

Whenever possible, parts A and B should be punched on separate pieces of tape. If the two must come on the same tape, they should be well separated by blank tape, and a note should be made under *Special Instructions* on the Operation Sheet.

It is essential to inform the operator of the presence of a precompile statement (OS 5), or if a precompiled tape is to be input before translation of a final segment (OS 4). It is possible to continue translation of the final segment and to run the program (OS 6) immediately after the initial segment has been precompiled.

Note. precompiled tapes prepared using a particular issue of the translator will normally work with subsequent issues. There may however be occasions when this is not true.

7. SUMMARY OF STANDARD PROCEDURES

abs	digits	instring	address	stop
sin	freepoint	outstring	size	wait
cos	scaled		range	restart
arctan	aligned	advance	lowbound	
entier	sameline	buffer		dump
sign	prefix		storemax	precompile
exp	reader			
ln	punch		elliott	checki
sqrt	grouping			checkr
tan	leadzero			checkb checkB
arcsin	special			checks
arccos				

Note. All the standard procedure names except checkr, checki, checkb, checkB, checks, elliott and precompile can be redeclared by the programmer. Within the scope of such a declaration they have the meaning given to them by the programmer; outside they have their standard meanings.

8. USE OF SAP COMMON PROGRAMS WITHIN AN ALGOL PROGRAM

If a SAP common program is read into store before the ALGOL system, then it can be used as a subroutine by an ALGOL program.

One method of entering the common program is as follows:

1. Include in the declarations of the ALGOL program, the declaration of the procedure 'enter CP'.

2. To achieve the same effect as the portion of SAP code:

```
COMP, CPname, n
  param 1
  param 2
    .
    .
    .
  param k.
```

write in the ALGOL program:

```
enterCP (CPinteger,n,NOPROG);
elliott (param 1)
elliott (param 2)
  .
  .
  .
  .
  .
elliott (param k)
```

the parameters being in pseudo-order form.

CPinteger is the integer representation of CPname (note that only the first six characters of the name are significant and that the name is left justified so that the binary representation contains 36 bits). NOPROG is a label to which a jump is made if the program CPname is not in store.

```
procedure enter CP(CPinteger,n,NOPROG);
value CPinteger,n;integer CPinteger,n; label NOPROG;

          elliott(0,0,0,0,3,0,CPinteger);
          elliott(7,3,7932,0,4,4,8082); comment RAPsearch;
ott(7.3. 4,1,4,2,9);
iott 2,0,CPinteger,0,0,0,0);
          ~~elliott(4,2,noprog,0,2,0,CPinteger)~~; comment get address of head;
          elliott(7,3,4,1,2,7,8189);
          elliott(2,0,4,1,7,7,8191); comment get procedure link;
          elliott(0,0,CPinteger,1,2,0,5); comment plant in LINK*CP;
          elliott(3,0,n,0,2,0,7914);
          elliott(0,0,CPinteger,1,3,0,1);
          elliott(5,1,20,0,2,0, CPinteger);
          elliott(0,0,CPinteger,1,0,0,3); comment jump into trigger block of CPname;
     goto NOPROG
end enterCP;
```

NOTES

(1) enterCP cannot be used if CPname expects to find its data in the accumulator. However, CPname can be modified to pick up this data from a workspace location whose address is given as the first parameter word. The call of CPname then takes the form

```
plant data in A;
     enterCP (CPinteger, n, NOPROG);
     elliott (0,0,0,0,0,0,A);
          .
          .
          .
```

(2) If NOPROG labels the statement "elliott (4,4,8064,0,0,0,0)", then the RAP message "NOPROG" is output if CPname is not in store.

(3) If CPname sets the overflow register, then the statement "elliott (7,3,4,1,4,3,1)" must be written immediately after the last parameter of CPname, so that overflow is cleared as soon as CPname is exited. Otherwise a spurious *Int Oflo* message occurs.

(4) CPname must be a proper SAP common program. That is to say, it must expect to find its link in LINKCP and exit must be made by an instruction EXITCP,n. CPname will then be stored in the manner described in chapter 2.1.2.6. of the 503 manual.

(5) There is not sufficient room in the store for SAP and ALGOL together. Therefore, CPname must be read in as a binary tape by the RAP instruction IN.

(6) More than one common program can be used by the same ALGOL program. All the common programs that are required must be read in *before* the ALGOL systems tapes.

Example

This example uses the library program PTSREAD as a common program. Its function is to read numbers from the typewriter, alternately **real** and **integer**, and to print each one out after it is read. If PTSREAD is not in store, the message *Try Again* is displayed.

When the program is entered, it uses the RAPread subroutine to obtain the integer equivalent of "PTSREAD" and assigns this to the integer variable PTSREAD. This is just a way of making the computer do the work of converting the alphanumeric into an integer.

Since both RAPread and PTSREAD can set the overflow, the "clear overflow" instruction appears after each entry.

```
Test enter CP;
begin integer PTSREAD,i; real A;switch S:=error,L;
comment insert enter CP here;
            elliott(7,3,7932,0,4,0,8020);
            elliott(7,3,4,1,4,3,1);
            elliott(3,0,7914,0,2,0,PTSREAD);
            comment enter RAP to read the name PTSREAD from the typewriter, and assign
              its integer representation to the variable PTSREAD;
                punch(3);
             L: enterCP(PTSREAD,1,error);
                elliott(0,0,0,0,0,0,4096);
                elliott(7,3,4,1,4,3,1);
                elliott(2,0,A,0,0,0,0); comment read real;
                print A;
                enterCP(PTSREAD,1,error);
                elliott(2,0,0,0,0,0,4096);
                elliott(7,3,4,1,4,3,1);
                elliott(2,0,i,0,0,0,0); comment read integer;
                print i;
                goto L; error:print ££l?Try again?
end;
```

CHAPTER 4

ERROR INDICATIONS

1. ERRORS DURING RUNNING OF PROGRAM

1.1. Overflow Errors

These errors prevent continuation of the program and are followed by a systems wait. The program is intact in the store, and can be re-run (see Chapter **5.2.4.**).

Displayed Message	*Error*
Int oflo	The result of some integer operation (e.g. division by zero) is outside the range -2^{38} to $2^{38}-1$.
Subscr oflo	A subscript of a subscripted variable is outside the range declared for the corresponding array or switch.
Space oflo	The program requires more space than is available in the computer.

1.2. Ignorable Errors

After one of these errors, the program can be continued by changing the leftmost F2 digit on the word generator.

x stands for the argument of the function; on continuation the value of the function is taken as zero.

Displayed Message	*Error*
Exp error	$x > 255 \log_e 2$ ($\simeq 176.75$)
Sine error (also for *cos*)	$\lvert x\rvert \geqslant 2^{30}$ ($\simeq 10^9$)
Log error	(*a*) $x \leqslant 0$ (*b*) attempted evaluation of $p \uparrow q$ with $p \leqslant 0$, and q real and $\leqslant 0$
Sqrt error	$x<0$
Arcsin error (also for *arccos*)	$\lvert x\rvert > 1$
Tan error	$\lvert x\rvert \geqslant 2^{30}$ ($\simeq 10^9$)
String error	The string being printed contains an inadmissible character. On continuation, the rest of the string is ignored.
Read error	(*a*) error found in the number being input (in particular, if a number expected to be an integer contains a decimal point or a subscript ten or is too large), or £ read at the start of a number. On continuation, the read subroutine is re-entered. (*b*) 'instring' has read either a numeric character before reading a £, or a £ within an inner string. On continuation, the program continues with the statement following the call of 'instring'.

Buffer error	The string parameter in a call of 'buffer' is an empty string. On continuation, the function is assigned the value **false.**
Print error	The real number to be printed is not in standardised floating-point form. On continuation, the printing of the number is omitted.

1.3. Machine Detected Errors

If the machine detects an error while the program is being run, the error indication 'ERRINT *n*' will be displayed. The significance of the values of *n* are given below:

ERRINT 1	Floating point overflow. Continuation is possible at programmer's discretion.
ERRINT 2	Occurs on a control shut-down due to a power failure, but there is insufficient time to display the message.
ERRINT 3	Parity error in the main store. Continuation not possible.
ERRINT 4	Error on peripheral equipment. Continuation not possible.
ERRINT 5	Impermissible reference to reserved area. Continuation not possible.

1.4. Undetectable errors

The following errors are not detected by the present compiler:

1. A formal parameter that is called by name has a value assigned to it in the procedure body, and the corresponding actual parameter is not a variable of the same declared type. In this case a dummy assignment is made.
2. An actual array parameter has a different number of dimensions or is of a different type from the corresponding formal parameter. The program is corrupt.
3. A label which has been placed twice in the same block. The first one only is recognised.
4. A label which has been declared but not placed. This may cause a jump to a wrong label or go completely wrong.

2. ERRORS DURING TRANSLATION OF PROGRAM

If an error is detected during translation, a message is displayed and a systems wait occurs. Translation cannot be continued but the remainder of the program can be checked for further syntactic errors.

The message has a heading of the form *Error no. n* where *n* is an integer giving a reference to the table in 2.1. of this chapter.

This is followed by a copy of the ALGOL text up to and including the next **end** or semicolon, and a systems wait occurs. If a **begin** is copied, then so is the corresponding **end.** If the error has occurred in the heading of a procedure declaration, then the whole body of the procedure is copied. In all cases the copied text is in no way checked for errors.

Error 20 (undeclared identifier) is treated specially. The name of the offending identifier is displayed, and translation continues, although no more owncode is produced. There is no copying of the text and no systems wait. The identifier is treated as if it had been declared in the head of the innermost current block to be of type **integer,** or, if the symbol immediately following it is an open bracket, of type **integer array.** If this declaration is incompatible with the use of the identifier—for example, if it is used as a label—then further, spurious, error indications will occur.

When the sign digit is changed after the systems wait, translation continues, although no more owncode is produced. At the end of translation, the name of the program is displayed and a dynamic stop occurs.

The FAME (Find ALGOL Mnemonic Errors) routine, which forms part of Tape 1, can also be used purely to check an ALGOL program for syntactic errors, without assembling any machine code. Such a check is quicker than a full translation, particularly if the program is so long that the own-code would have to be output onto paper tape.

To use this facility the following instructions should be obeyed:

(a) If the first part of the program is a precompiled tape, read this in by typing IN. Then, or otherwise

(b) Load the ALGOL program tape in the reader and type FAME.

At the beginning and end of the checking a few characters will be output on Punch 1. These should be ignored. Error messages and the name of the program appear on the output writer.

If the statement precompile is read after an error has been found in a program or after typing FAME., the message *Error no.* 53 is output. If the sign of the word generator is changed, checking continues. Spurious error messages are produced by the title of the second part of the program.

The explanations given by the error table will normally be correct. There are, however, some cases when the translation will detect that there is an error in the program, but will attribute it to a cause apparently totally unrelated to what has been written. When this happens, the actual error will usually be found in a statement near the point where translation has stopped.

The FAME system cannot be foolproof. Although it is unlikely that an actual error will pass undetected (except in text being copied), it is possible for one actual error to lead to several spurious error indications, and even to complete breakdown. If the error indications begin to come in nearly every statement, the operator would be advised to remove the ALGOL program from the machine and to carry on with the next one.

Errors in declarations are bound to cause spurious error indications later since part of the declaration will be skipped: but if all the errors in a program occur in statements, there is a good probability that all of them will be detected correctly. However, the first error found is a real one.

It is important to mark the tape on the right of the central perspex strip when an error occurs, because the amount of text copied may be insufficient to identify the position of the error.

2.1. Error Table

n	*Error*
1 (a)	Number of impermissible form.
(b)	Error in an underlined word.
2 (a)	Impermissible beginning to a statement.
(b)	Symbol following **own** not **real, integer, Boolean** or **array.**
(c)	Procedure declaration not terminated by semicolon.
(d)	Name in declaration not followed by comma or semicolon.
(e)	Declaration following procedure declaration not a procedure declaration.
3	Name declared twice in same block head.
4	Item after a comma in a declaration not a name.
5 } 6 }	First identifier in switch declaration not followed by ':='.
7	In a procedure declaration:
(a)	Item following procedure name not semicolon or open bracket.
(b)	No semicolon or close bracket after formal parameter part. **N.B.** If the item following the close bracket is not a semicolon, the compiler will ignore a letter string until ':'.
(c)	List in value or specification part has impermissible form.
(d)	Specification part occurs before value part.
8 (a)	Parameter of non-allowable type.

n	*Error*
(b)	Too many parameters (> 1023).
(c)	Parameter not specified.
9	Recursive procedure with **real, integer** or **Boolean** name parameter.
10	Name in value part not a formal parameter.
11 (a)	Name in specification part not a formal parameter.
(b)	Parameter specified twice.
12	**go to** statement leading out of a procedure body.
13	In array declaration:
(a)	No comma or open bracket after identifier.
(b)	No colon between upper and lower bounds.
(c)	No comma or close bracket after bound pair.
14	Negative **own array** size.
15	**own array** with variable bounds.
16	**array** with more than 31 dimensions.
17 (a)	No **end** or semicolon after statement in compound tail.
(b)	No colon after a label.
(c)	Impermissible beginning to a statement.
18	Left part variable in assignment statement not followed by ':='.
19	Value assigned to a procedure identifier outside procedure body.
20	Identifier not declared, or used outside scope of declaration.
21	'location' not followed by '[integer identifier]'.
22	Inadmissible complex primary in arithmetic expression.
23	Empty arithmetic expression.
24	Missing operand in arithmetic expression.
25	No close bracket after subscript list.
26	Missing bracket.
27	Wrong number of subscripts in subscripted variable.
28	Missing **else**.
29	Missing **then**.
30	Conditional statement or expression after **then**.
31 (a)	Missing or inadmissible operator in arithmetic expression.
(b)	Boolean operand in arithmetic expression.
(c)	Missing open bracket after name of procedure with parameters.
32	Non-identifier or Boolean variable in a **read** list.
33 (a)	Inadmissible identifier as left part element.
(b)	Inadmissible identifier as controlled variable in **for** statement.
34	Empty **Boolean** expression.
35	Missing relational operator.
36	Missing operand in **Boolean** expression.
37	Inadmissible complex Boolean primary.
38	Inadmissible operator in **Boolean** expression.
39	Type procedure name in **read** list, but not within its own body.
40	Inadmissible symbol at start of an expression.
41	During procedure call:
(a)	No open bracket following name of procedure with parameters.
(b)	Actual parameter not followed by comma or close bracket (cf. error 7 (b)).
42	Error in parameter delimiter of form ') letter string:('.
43	No actual array or string parameter where one expected in a procedure call.

n	*Error*
44	Non-allowable parameter in a procedure call.
45 (a)	Controlled variable in for statement not followed by ':='.
(b)	For list element not followed by a comma or **do**.
46	Incorrect designational expression.
47	Arithmetic expression in **for** list element not followed by **step**, **while**, **do** or a comma.
48	Missing **until**.
49	Program too large or complex to be compiled at all.
50	Label not declared in innermost possible block.
51	Error in a call of procedure 'elliott'.
52	Issue number of compiler current when program was *precompiled*, not same as one now being used.
53 (a)	'precompile' not followed by semicolon.
(b)	Occurrence of 'precompile', in program being translated to paper tape.
(c)	Attempt to produce or use a precompiled tape when the ALGOL system is not in the same position in the store as it was when the tape was prepared.
54 (a)	Occurrence of a £ within an inner string (possibly caused by omission of ? at end of a previous string).
(b)	**comment** occurs otherwise than after a semicolon or a **begin**.

N.B. The following errors will cause an ALGOL program tape to shoot through the reader, instead of stopping at the end:

(i) No 'halt' sign at the end of a tape which is not the last tape of the program.
(ii) Insufficient **end**'s to match all the **begin**'s in the program.
(iii) No semicolon after the final **end**.
(iv) Missing ? at the end of a string, causing the program statements that follow to be treated as part of the string. (This error will be detected (Error 54) if an inner string occurs in any subsequent statement.)

See Appendix II for notes on program checking and common errors made in program writing.

CHAPTER 5

OPERATION OF THE ALGOL SYSTEM

OPERATING NOTES

2.1. Introduction

This chapter describes the operation of the Elliott ALGOL system on the 503.

The description falls into two parts:

The first is a general introduction to the Normal Operating System, Additional Facilities and the use of the Operation Sheet. It will not teach the reader how to operate the system, but merely give him a general idea of how it behaves.

The second part contains the actual operating instructions. These are intended for reference when operating the system on a machine, and are set out in tabular form, with the operator's *actions* separated from the description of what happens.

2.2. Normal Operation

The ALGOL system on the 503 is used under the control of RAP i.e. RAP must be in the store before an ALGOL program can be either translated or run.

In Normal Operation of the ALGOL System on the 503, ALGOL programs are run one after the other ('load and go'). First the ALGOL tapes 1 and 2 (the ALGOL systems tapes) are input and then each program is translated into the store of the computer, and is run immediately after translation is complete. As soon as the running of one program has finished, the translation of the next program takes place.

The ALGOL System normally remains in the computer throughout the translation and running of a whole batch of short programs. If, however, one of the programs requires a large amount of space, the ALGOL system is overwritten in store, and has to be reinput before translation of the next program can begin.

The messages, such as ALGOL or OWNIN, which the operator types in order to control the system, are read by RAP and so must always be terminated by a full stop.

Systems and Data Waits

When the computer waits for action from the operator, two types of situation are distinguished.

In a *systems wait*, the ALGOL translator has control, and in general the operator is expected to place an ALGOL program tape in the reader. Translation will continue when the setting of the word generator sign digit is changed.

In a *data wait*, the ALGOL program currently in the store has control, and in general the operator is expected to place a data tape in one of the readers. Calculation will continue when the operator changes the setting of the leftmost F2 digit.

To achieve any other effect during a systems or a data wait, an entry has to be made via RAP.

When a systems or data wait occurs, an appropriate *Dwait* or *Swait* message is displayed.

The Translation Stage

The translation of an ALGOL program into the store of the computer takes place in two stages. During the first stage, the complete ALGOL program is read in, and at the end the title of the program is displayed.

The second stage is purely internal, and may last a few seconds. Finally a *Free store* message is displayed, and the program is in the store, ready to be obeyed (see Chapter **3.5.1.**).

Normal Input Channel

The ALGOL tapes 1 and 2, and ALGOL program tapes, are input by the reader connected to channel 1. In operating instructions such as 'place tape 1 in reader', it is this reader that is meant. An operator can save time by putting tapes in both readers, and manipulating the reader select button.

More detailed information on Normal Operation may be found on page 40.

The Operation Sheet

Whenever an operator is to run an ALGOL program, he should be given clear and detailed instructions concerning what is to be done. These can conveniently be set out on an Operation Sheet of the form illustrated (see fold-out sheet at the end of this section).

If no departure from Normal Operation is required, only the first section of the sheet (above the double line) need be filled in. If any Additional Operations are required, these are indicated by placing a tick or a number in the appropriate box. If there are any further instructions to the operator, not covered by the entries provided on the Operation Sheet, then these should be written in the space marked 'SPECIAL INSTRUCTIONS'. If any of these further instructions relate to a particular item of the Operation Sheet, this item should be marked with an asterisk.

The Operation Sheet should also be used by the operator to convey information to the programmer in cases where some kind of error condition arises. The space on the right hand side of the sheet is provided for this purpose.

In what follows, OS *n* refers to item *n* of the Operation Sheet.

2.3. Error Conditions in Normal Operation

During Translation

If an error is found during the translation stage, a message headed *Error no. n* is displayed, followed by a systems wait.

The operator should preserve the error message, and begin translation of the next program by changing the sign digit; or mark the tape and change the sign of the word generator to check for further errors.

During Running

If an ALGOL program contains an error not detected during translation, or if it makes demands beyond the capacity of the computer, an error condition arises which, in general, prevents continuation of the program. In such a case the operator is expected to make the appropriate comment on the Operation Sheet, and to proceed to the next program. However, in certain circumstances the programmer can ask for continuation, and should tick the appropriate box of OS 15. If OS 15 is not marked, it is assumed that the program is not to be continued.

In cases during the running of a program where continuation is never possible, an *Oflo* message is displayed followed by a systems wait: viz. *Int oflo*, *Subscr oflo*, *Space oflo*. To start translation of the next program, change the sign digit.

In the case of *Space oflo*, the ALGOL system is overwritten in the store and must be reinput before translation can start.

In cases during the running of a program where continuation is permitted, an *Error* message is displayed followed by a data wait: e.g. *Sine error*, *Exp error*. To continue running the same program, change the leftmost F2 digit. If continuation is not required, translation of the next program can be started only by pressing the MESSAGE button and then typing ALGOL.

More detailed information on Normal Operation and on Error Conditions may be found on pages 40 and 41.

2.4. Additional Facilities

Checks

The 'checking' facility (OS 2 and 9) provides optional output during the running of the program. If these items are marked, the B-digit must be depressed throughout translation and/or running.

N.B. Make sure that the B-digit is raised in all cases where checking is not required.

Program and Data on More than One Tape

When a program is punched on more than one tape, the computer will come to a systems wait at the end of each tape, and the operator is expected to put the next tape in the reader. When data is punched on more than one tape, it is up to the programmer to specify how these tapes are to be used (SPECIAL INSTRUCTIONS).

When more than one tape is used, the programmer must mark OS 3 or 10, and the operator should check that the required number of tapes is ready, and that their sequence is known, before starting to run the program. If the number of data tapes is marked as zero, this means *either* that no data is required *or* that the data has been punched at the end of the program tape. In this latter case the program tape will not be read to the end at the translation stage, and should be left in the reader when the running of the program is started.

Use of Equipment

If a second reader or punch is required during the running of the program, OS 8 should be marked. This will warn the operator not to attempt to run the program on a computer on which the relevant equipment is missing or out of order.

Wait

When the program itself contains an instruction to cause the computer to wait, a data wait will occur. The programmer must indicate whether this kind of wait is to be expected (OS 12) and what action is required (SPECIAL INSTRUCTIONS).

Manual Restart

If the programmer wishes to indicate that the program is to be rerun (for example, with a different set of data), he must tick OS 16, and also note under SPECIAL INSTRUCTIONS which data tape is to be input. Before continuing with the next program, the operator should always make sure that the current program is not to be run again via a manual restart.

A manual restart may also be used to begin the program again after a false start.

2.5. Advanced Operations

Advanced Operations are concerned with methods of using ALGOL which differ from the 'load and go' system described in the previous sections. Instead of translating the program into the store, a version of the program is produced on paper tape for subsequent input and use, or the text is checked for errors but is not translated.

There are three different ways of producing and using these paper tape versions of the program:

1. The Precompiled Package.
2. The Owncode Program.
3. The Dumped Program.

Precompiled Package

A precompiled package is a portion of ALGOL program which is processed in advance so that it may be used as the first part of some other program. It can be used only when the ALGOL system is in the same position in the store as it was when the package was produced.

Such a package will generally represent a set of one or more commonly used procedures, which can be used again and again by different programs without the need to retranslate them each time.

A precompiled package is produced as a result of writing the appropriate instruction in the program being translated. It is a by-product of the translation, and the translation and running of the program which produced it may continue afterwards in the normal way (OS 6).

The package is used at the translation stage as the first tape of an ALGOL program; second and subsequent program tapes must be normal ALGOL tapes, and are translated immediately after the input of the package.

Owncode Programs

The two stages of the translation of an ALGOL program can be separated. An intermediate version of the program—called owncode—is produced on paper tape for reinput later, possibly on another computer (503 or 803). Translation to owncode is limited by the speed of the output punch and is much slower than normal translation.

If an ALGOL program is too long to fit into the store together with the translator, an owncode version is produced automatically; this should be run in the same way as a program which has been deliberately translated to owncode. An owncode program can be produced, either deliberately or automatically, at any time without affecting Normal Operation of the system.

Note: Owncode program tapes must be wound up *backwards;* this means that the end *first output* is placed on the winding spool first.

An owncode program cannot be *run* during Normal Operation. The operator should save all owncode programs to be run later—both those produced at the current Normal Session and those given in for running only (OS 7b).

When owncode programs are to be run, tape 2 (but not tape 1) must first be input. Owncode programs can then be input and obeyed one after the other. Only if a program has required the whole store is it necessary to reinput tape 2 before running the next program. In this case, the message '*Reinput tape* 2' will be displayed.

Dumped Programs

A dumped tape is a binary version of a complete program, suitable for input by RAP. It contains all the parts of the ALGOL system that are necessary for its running.

A dumped version may be produced during the running of a program by an instruction in the program itself (OS 13). Alternatively it may be produced manually by the operator immediately before the program is run (OS 11). Dumped programs may be run at any time without previously reading in either of the ALGOL system tapes; they are reinput into the same area of the store from which they were coded out. If the dumped program uses any programs external to the ALGOL system, these will not have been coded out and must be input before the dumped program tape.

Producing Precompiled Packages and Dumped Programs

Precompiled packages and dumped programs are binary tapes which are produced and checked on a time-shared basis by the following procedure:

(1) Output of the binary tape begins on punch 1.

(2) When the tape produced is long enough to reach the tape reader, the leading end is placed in reader 1 and the keyboard sign digit is changed. This causes checking character by character to be carried out simultaneously with the remaining output.

(3) When output has finished, the tape is run out and torn off and the sign digit is changed. This causes the final section of the tape to be checked.

The display of START CHECK during stage 1 means that the store available for holding the tape words to be checked is now full, and the operator must take one of the following actions:

(1) If the tape is long enough, proceed as in (2) and (3) above.

(2) If the tape produced so far will not reach the reader:

  (*a*) change the *B-digit* of the word generator. (This causes the output to continue.)

  (*b*) When output has finished, run out the tape and tear off. To check the tape, load the tape reader and change the sign digit. The tape will then be read and a sum check carried out.

In either case, the display of PUNCH ERROR means that the whole process must be restarted by another message through RAP.

2.6. Marking of Tapes

It is suggested that the following conventions be used in marking tapes:

Tape	*Marking*
(i) *The Programmer's Responsibility*	
Program tapes	PR1, PR2, etc.
Data tapes for reader 1	DT1/R1, DT2/R1, etc.
Data Tapes for reader 2	DT1/R2, DT2/R2, etc.

(ii) *The Operator's Responsibility*

Results tapes from punch 1	RT1/P1, RT2/P1, etc.
Results tapes from punch 2	RT1/P2, RT2/P2, etc.
An owncode program	OC
A precompiled package	PP
A dumped program	DP

In addition, all tapes must be labelled with the title of the program.

3. OPERATING INSTRUCTIONS

3.1. Normal Operating Instructions

	Action	*Result*
Stage 1	Place systems tape 1 in reader; **press the MESSAGE button and type IN.**	The tape is read in and stops at the end in a *systems wait.*
Stage 2	Place systems tape 2 in reader; **change sign digit.**	The tape is read in and stops at the end in a *systems wait.*
Stage 3 (translation)	Place ALGOL program in reader; **change sign digit.**	The tape is read at variable speed to the end. The title of the program is displayed, followed by a *Free store* message and a *data wait.* Check that the title is the same as that on the Operation Sheet.
Stage 4 (running)	Place data tapes in readers as specified by programmer (OS 10); **change leftmost F2 digit.**	The program is now run. At the end, the message *End of program* is displayed and a *systems wait* occurs.
	If any further ALGOL programs remain to be processed, go back to Stage 3.	

3.2. Error Conditions in Normal Operation

	Condition	*Action*
During Stage 1	1. ERRSUM is displayed.	Repeat Stage 1.
During Stage 2	2. ERRSUM is displayed.	Replace tape 2 in reader, **type** REIN 2.
During Stage 3 (translation)	3. The ALGOL program is partially read; a message headed *Error no. n* is displayed; a *systems wait* occurs.	Remove offending program, save displayed message, and go on to next program, or mark the tape see (3.1.4.2.) and change the sign of the word generator to find further errors.
	4. During translation of a long program, a sudden stream of output occurs on punch 1, lasting for several minutes, followed by alternating input and output, and ending in a *systems wait.*	The tape just output (which is an owncode version of the program) should be wound up *backwards* and run later—in the same way as other owncode programs (p. 45). To continue normal translation of next program, **change sign digit.**
	5. Continuous output of binary 1 or 2 after the title of the program and before the *Free store* message has been displayed, indicating a machine error.	Retranslate the program by **pressing the** MESSAGE **button and typing** ALGOL.

	Condition	*Action*
During Stage 4 (running)	6. An overflow message *Int oflo* or *Subscr oflo* is displayed, followed by a *systems wait.*	Go on to next program at Stage 3.
	7. An overflow message *Space oflo* *Reinput tapes 1 and 2* is displayed.	Clear main store by typing either RESET. or CANCEL; ALGOL 1. and then go on to next program at stage 1.
	8. ERRINT 1 is displayed.	Floating point overflow has occurred. Note this under operator's comments on the Operation Sheet. If OS 15a is ticked, **type** CONT; ERRINT. Otherwise go on to next program by **pressing the** MESSAGE **button and typing** ALGOL.
	9. An error message is displayed followed by a *data wait* (e.g. *Sine error*).	If OS 15b is ticked, **change leftmost F2 digit**; otherwise go on to next program by **pressing the** MESSAGE **button and typing** ALGOL.
	10. The program exceeds the maximum time declared by the programmer.	Go on to next program by **pressing the** MESSAGE **button and typing** ALGOL.
	11. *Reinput tapes 1 and 2* is displayed (i) after the message *End of program* has been displayed or, (ii) after typing ALGOL.	To continue with next program, clear main store by typing either RESET. or CANCEL; ALGOL 1. and then go on to next program at stage 1.

3.3. Operating Instructions for Additional and Advanced Procedures

(a) During Translation

	Condition	*Action*	*Result*
OS 1b* translation to owncode	—	(i) To start translation place program tape in reader; **press the** MESSAGE **button and type** OWNOUT.	An owncode tape of the program is produced on punch 1. At the end the computer waits for a further message. For errors see Error Conditions in Normal Operation (Stage 3).
		(ii) Wind the tape up *backwards*, and mark it with the title of the program. (iii) (*a*) To continue normal translation of next program, **type** ALGOL. (*b*) To continue translation to owncode, repeat from (i) above.	
OS 1c	No translation required	See instructions for RUNNING.	—
OS 1d syntactic check	—	To start checking, place the program tape in the reader, **press the** MESSAGE **button and type** FAME.	The program is checked for syntactic errors. At the end the computer waits for a further message. For errors see Error Conditions in Normal Operation (Stage 3).
OS 2	—	Before starting translation, **depress the word generator B-digit.**	—

* N.B. If at a given session it is required only to translate programs to owncode, then Stage 2 of the Normal Operating Instructions may be omitted (i.e., tape 2 need not be input).

	Condition	*Action*	*Result*
OS 3	At the end of each program tape but the last, input stops and a *systems wait* occurs.	Place next program tape in reader; **change sign digit.**	Translation continues as before.
OS 4	The first program tape is a precompiled package.	(i) Place precompiled tape in reader; **type** IN.	The tape is read in and stops at the end in a *systems wait*. For errors see p. 48.
		(ii) Place next tape of the same program in reader; **change sign digit,** or if syntactic checking only is required **type** FAME.	Translation or checking continues; at the end the title of the pre-compiled package is displayed together with the title of the main program.
OS 5	At the end of input of a program tape, output of the precompiled package starts on punch 1.	(i) When enough tape has been output to reach comfortably to the reader, **depress the** MANUAL **button** on the control panel. Place the end of the precompiled tape in the reader, **change the sign digit, and release the** MANUAL **button.**	(i) Simultaneous output and checking of the precompiled package takes place. For errors see p. 48.
		(ii) When output stops, run-out tape and tear off. Then **change the sign digit.**	(ii) The final section of the tape is checked, and the computer waits for a further message.
		(iii) If OS 6 is ticked, continue translation of the same program by placing next program tape in reader, and **typing** PRE 2; otherwise, to go on to normal translation of next program, **type** ALGOL.	

Operating Instructions for Additional and Advanced Procedures

(b) During Running

	Condition	*Action*	*Result*
OS 7b Running owncode programs.	—	(i) Place systems tape 2 in reader; **press the MESSAGE button and type** IN.	The tape is input and stops at the end in a *systems wait*. For errors see p. 48.
		(ii) Place the owncode program in reader; **change sign digit.**	The title of the program is displayed, and the tape is input. The *Free store* message is displayed, followed by a *data wait*. For errors see p. 48.
		(iii) Place data tapes in readers as specified by programmer (OS 10); **change leftmost F2 digit.**	The program is now run. At the end, the message *End of program* is displayed, followed by a *systems wait*. For errors, see Error Conditions in Normal Operation, Stage 4.
		(iv) If any other owncode programs remain to be run, go back to (ii) above.	

	Condition	*Action*	*Result*
OS 7c Running dumped programs.	—	(i) Place the program tape in reader; **press the MESSAGE button and type IN.**	The tape is input and stops at the end in a *data wait.* For errors see p. 48.
		(ii) Place data tapes in readers as specified by programmer (OS 10); **change leftmost F2 digit.**	The program is now run. At the end *End of program* is displayed. For errors see Error Conditions in Normal Operation, Stage 4.
OS 8	The specified devices are required by the program.	Check that these devices are available and supplied with tape, paper, ribbon, etc.	—
OS 9	—	Before starting to run the program, **depress the B-digit.**	Checks are output.
OS 10	The specified number of data tapes are used by the program.	Check that all the data tapes are available, and that their order of input is known.	—
OS 11	—	(i) Before starting to run the program, **press the MESSAGE button and type** DUMP 1.	(i) Output of the dumped program starts on punch 1.
		(ii) As soon as enough tape has been output to reach comfortably to the reader, **depress the MANUAL button** on the control panel, place the end of the dumped tape in the reader, **change the sign digit and release the MANUAL button.**	(ii) Simultaneous output and checking of the dumped tape takes place. For errors see p. 48.
		(iii) When output stops, run-out tape and tear off. Then **change the sign digit.**	(iii) The final section of the tape is checked, and the computer waits for a further message.
		(iv) If OS 14 is ticked see below; otherwise go on to next program by **typing ALGOL.**	

	Condition	Action	Result
OS 12	The program stops in a *data wait*.	See OS SPECIAL INSTRUCTIONS.	
OS 13	Output of a dumped tape begins on punch 1.	(i) Proceed as for (ii), (iii), (iv) of OS 11	The dumped program is output and checked (for errors see p. 48).
OS 14	—	Place data tapes in readers as specified by programmer; **type** DUMP 2.	The program continues running.
OS 15	The program stops having displayed an *Oflo*, *Error*, or ERRINT message.	See Error Conditions in Normal Operation (items 6, 7, 8, 9) p. 42.	
OS 16	—	Place data tapes in readers as specified by programmer; **press the** MESSAGE **button and type** REPEAT.	The program is run again.

3.4. Error Conditions during Additional and Advanced Procedures

(a) During Translation

	Condition	*Action*
OS 4 Input of precompiled package.	(i) ERRSUM is displayed indicating a sumcheck error.	(i) Re-input precompiled package by **typing** IN.
	(ii) *Error no.* 53 is displayed.	(ii) Carry on with next program.
OS 5 Precompiled package output.	(i) PUNCH ERROR is displayed.	(i) Re-output precompiled package by **typing** PRE **3.**
	(ii) START CHECK is displayed.	(ii) (*a*) If the tape is long enough to reach the tape reader, load the reader and **change the sign digit.** (*b*) If the tape is not long enough to reach the reader, depress the B-digit, causing output to continue. When the punch has finished its output, check the tape by loading it in the reader if necessary and **changing the sign digit.**
	(iii) *Error no* 53 is displayed.	(iii) Carry on with the next program or mark the tape and change the sign of the word generator to find further errors.

(b) During Running

	Condition	*Action*
OS 7b (i) Running owncode programs.	ERRSUM is displayed indicating checksum error.	Re-input systems tape 2 by **typing** IN.
(ii)	Continuous output of binary 1 or 2 indicating a wrong character or a sumcheck error.	Replace owncode program tape in reader, **press the** MESSAGE **button and type** OWNIN. If it still fails the ALGOL program will have to be translated again at another session.
	Display of the message *Space oflo* indicating the program is too large to fit in the store.	Nothing can be done.
(iii)	Error conditions are as for Normal Operation, except that *Reinput tape 2* is output in place of *Reinput tapes 1 and 2*.	As for Normal Operation, except that 'go on to next program by **typing** ALGOL' is replaced by 'input next program by **typing** OWNIN.'
OS 7c Running dumped programs.	ERRSUM is displayed indicating a sumcheck error.	Re-input dumped program tape by **typing** IN.

OS 11 (ii) **OS 13 (i)**	PUNCH ERROR is displayed.	Re-output dumped program by **typing** DUMP 3.

4. SUMMARIES

4.1. Summary of Waits

Systems Wait

Condition	*Action*
1. After input of systems tape 1 in Normal Operation.	Place systems tape 2 in reader.
2. After input of systems tape 2.	Place ALGOL program tape in reader.
3. If an ALGOL program is punched on several tapes, a systems wait will occur at the end of each tape except the last.	Place next tape of the program in reader.
4. After end of execution of an ALGOL program; or after an irretrievable error during running or a syntactic error during translation.	Place next ALGOL program in reader; or, during translation, change the sign of the word generator to check for further errors.

Data Wait

1. After the input of an ALGOL program.	Place data tape(s) in readers as specified by programmer (OS 10).
2. During running of ALGOL program, due to WAIT in program or 'stop' on data tape.	Generally, place next data tape in reader as specified by programmer (OS 10, 12, SPECIAL INSTRUCTIONS).
3. Retrievable error stop, e.g. *Read error.*	Continue only if OS 15b ticked.

Error Wait

Floating point overflow.	Continue only if OS 15a ticked.

RAP Wait

1. After display	
Reinput tapes 1 and 2	Reinput tapes 1 and 2.
Reinput tape 2	Reinput tape 2.
2. After outputting and checking precompiled package.	*Either* continue translation by **typing** PRE 2 (OS 6), *or* translate next program by **typing** ALGOL.
3. After outputting and checking a dumped program.	*Either* continue running program by **typing** DUMP 2 (OS 14), *or* translate next program by **typing** ALGOL.

2.1.3.5.

4.2. Summary of Input Messages

1. *During Normal Running*

IN.	(*a*) To input systems tape 1.
	(*b*) To input precompiled program.
OWNOUT.	To translate an ALGOL program to owncode (OS 1b).
REPEAT.	To restart manually the program currently in the store (OS 16).
ALGOL.	To translate an ALGOL program normally.
DUMP 1.	To dump manually the program just input (OS 11).
DUMP 2.	To continue running a program after it has been dumped (OS 14).
DUMP 3.	To re-output a dumped program after a punch error.
PRE 2.	To continue translating a program after the first part has been precompiled (OS 6).
PRE 3.	To re-output a precompiled program after a punch error.
REIN 2.	To input systems tape 2 after sumcheck failure.
FAME.	To check for syntactic errors without assembling the program.

2. *During Running of an Owncode Program*

IN.	Input systems tape 2.
OWNIN.	Input an owncode program.

REPEAT; DUMP 1; DUMP 2; DUMP 3—as during normal running.

4.3. Summary of Suggested Tape Titles

Identification on tape	*Meaning*
DP *title*	dumped program
DT*n*/R1 *title*	*n*th data tape for reader 1
DT*n*/R2 *title*	*n*th data tape for reader 2
PP *title*	precompiled package
PR*n* *title*	*n*th program tape
RT*n*/P1 *title*	*n*th results tape from punch 1
RT*n*/P2 *title*	*n*th results tape from punch 2
OC *title*	owncode program

4.4. Summary of Displayed Messages

During Translation	
Error no. n	program can be checked for further errors.
Title of program	
Free store = n to m	program ready to run
During running	
Exp error	
Sine error	
Log error	
Sqrt error	
Arcsin error	
Tan error	continuation at programmer's discretion (OS 15a, b)
String error	
Print error	
Read error	
Buffer error	
ERRINT 1	
Int oflo	
Subscr oflo	continuation *not* possible
Space oflo	
ERRINT (2—5)	
End of program	
Reinput tapes 1 *and* 2	
Reinput tape 2	
PUNCH ERROR	
START CHECK	
ERRSUM	
Swait	
Dwait	

C. A. R. Hoare, J. S. Hillmore, R. Grover, S. Gilbert *January 1965*

APPENDIX I

The 503 ALGOL System for Machines with a Core Backing Store

1. The systems tapes 1 and 2 can be kept on core backing store, and can be read down into the main store ready for normal translation and running of ALGOL programs, by just inputting a leader tape.

N.B. Owncode programs cannot be run using the ALGOL system from the backing store.

2. When an ALGOL program is too large to be translated normally in the store, instead of an owncode version being output on paper tape, the owncode is written onto the backing store. This allows the large program still to be run load and go, although the translator is overwritten in the store.

3. OPERATING INSTRUCTIONS to write the ALGOL system to CORE BACKING STORE

3.1. Normal Operating Instructions

	Action	*Result*
Stage 1	Place systems tape 1 in reader; **press the** MESSAGE **button and type** IN.	The tape is read in and stops at the end in a *systems wait*.
Stage 2	Place systems tape 3 (ALGOL to backing store) in reader; **press the** MESSAGE **button and type** IN.	The tape is read in and END is displayed.
Stage 3	**Type** STORE.	Tape 1 is written on the core backing store and REIN 2 is displayed.
	Place systems tape 2 in the reader and **change the sign digit.**	Tape 2 is read into the store and ALGOL 2 is displayed.
Stage 4	**Press the** MESSAGE **button and type** ALGOL 3. **followed by** TAPE 2.	Tape 2 is written onto the core backing store and the leader tape is output on punch 1.
	To check the leader tape for punching errors, run out some blanks, tear off the output, place the leading end in the tape reader and **change the sign digit.**	If the leader tape is correct END is displayed.

3.2. Error Conditions in Normal Operation

	Condition	*Action*
During Stage 1	ERRSUM is displayed.	Repeat Stage 1.
During Stage 2	ERRSUM is displayed.	Repeat Stage 2.
At the end of Stage 3	ERRSUM is displayed.	Replace tape 2 in the reader and **type** REIN 2.
At the end of Stage 4	*errsum* is displayed	Discard the leader tape and **type** CODE. to output another leader tape. This can be checked as in Stage 4.

3.3. Bringing Down the Systems Tapes into the Main Store

	Action	*Result*
Stage 1	Place the leader tape in the reader and **type** IN.	LEADERUNCHEK is displayed and then END and *Swait*. Until this happens, no other tapes must be input.
	To translate an ALGOL program place the tape in the reader and **change the sign of the word generator.**	
Stage 2	The leader tape remains in the store all the time so that provided the systems tapes have been read down correctly once, they can be read down again by **typing** LEADER. To bring tape 2 into the same position as before **type** CANCEL; ~~TAPE 2.~~ALGOL 2. before **typing** LEADER.	Tape 1 is read into the same position as before, but tape 2 is read into a new position depending on the value of the RAP L.F. pointer.

3.4. The Position of the ALGOL System on the Backing Store

The normal operating procedure causes the ALGOL system to be written into location 0 and upwards of the core backing store. If it is desired to store the system elsewhere, say from location N upwards, then insert the following step between Stages 2 and 3 of Normal Operating:—

	Punch on a data tape the integer N terminated by a newline. Put the data tape in the reader and **type** ADDR.	The data tape is read in, and END is displayed. If error is displayed, there is a mistake on the data tape; put a corrected tape in the reader and **type** ADDR. again.

Note. During the process of writing tape 2 onto the backing store, tape 1 in the main store is overwritten and is removed from the RAP list. Therefore, to rewrite systems tapes 1 and 2 onto core backing store, the whole process must be repeated from the beginning,
i.e. **type** RESET. to clear the main store and reset the RAP pointers, and then proceed from stage 1.

3.5. Summary of Error Indications

(a) During writing up

Message	*Meaning and Action*
ERRSUM	The tape checksum has failed. Reinput the tape.
error	(i) Incorrect entry message. e.g., an integer, impermissible word or a terminating digit other than '.' (full stop). **Type the message again.** (ii) Incorrect address on ADDR. data tape e.g., the number contains an impermissible character or the number is out of range, i.e., the address does not exist on the units of core backing store present. Place a corrected data tape in the tape reader and **type** ADDR. again.
△	Impermissible character in the message, **type the word again.**
P	Backing store parity occurred when reading a block for checking. The incorrect block is rewritten and rechecked. if it is then correct, C is displayed; otherwise P is displayed again.
E	A check block's checksum has failed. Further action is as for P, except that if the checksum fails again E is displayed.
C	An incorrect block has now been written correctly. The program continues.
A string of P's or E's	The same block is failing all this time which probably means there is something wrong with the backing store.
A string of C's	This could mean that ALGOL tape 3 has been overwritten. Repeat the whole process from the beginning.
NOROOM	The free store is not large enough to hold tapes 1, 2 and 3. Cancel some programs and start again.
errsum	The leader tape is incorrect. **Type** CODE and a new copy of the leader tape is output.

(b) During reading down

Message	*Meaning and Action*
wrong leader	The issue numbers on the leader tape do not correspond with the issue numbers of the systems tapes on the core backing store. Start again with the correct leader tape.
bspar	Backing store parity has occurred during the reading down process and the ALGOL system is probably corrupt. A marker is set on the backing store and any attempt to read down the corrupted ALGOL system causes NOPROG to be displayed.
NOPROG	The ALGOL system has been corrupted and must be rewritten onto the backing store.
ERRSUM	Either the leader tape's checksum or the ALGOL system's checksum has failed. If the former, reinput the leader tape; otherwise rewrite the systems tapes onto the backing store.
NOROOM	The free store is not large enough to hold the leader tape and the ALGOL system. **Type** RESET. and reinput the leader tape.

Control is transferred to RAP after all these error indications.

3.6. The Program is too Large for Normal Operating

(i) If a program is too large to be translated normally into the store then it is written onto the backing store automatically. Before the transfer of owncode to the backing store begins, a message SET UP BSLIMS is displayed, followed by *Swait*. If there are parts of the backing store which may not be written into, then set up the limits of the free area of the backing store on the word generator before continuing. The lower limit should be set on the F1 N1 buttons and the upper limit on the F2 N2 buttons. The assumed free area of the backing store is locations 8192 to 16,383 inclusive (i.e., allowing for the ALGOL system in the standard position). If this area is free, then clear the word generator buttons before continuing. In other respects the operation of the program is exactly the same as if it were being run normally.

If *error* is displayed, followed by *Swait*, then either

(a) the lower limit $>$ the upper limit -64

or (b) the address defined by the upper limit is non-existent.

Set the limits up correctly and continue by clearing the systems wait.

(ii) When the free area of the backing store is full a message *bsfull* is displayed followed by a systems wait. Translation of the current program cannot be continued. To start translating the next program, **change the sign of the word generator.**

(iii) If *bspar* is displayed, followed by a systems wait, then backing store parity error has occurred and translation cannot continue. **Change the sign of the word generator** to translate the next program.

Note

When a program is too large to be translated normally, the owncode is written into the backing store only if the ALGOL system has been read from backing store. If ALGOL has been input from paper tape, the backing store is not used.

APPENDIX II

COMMON ERRORS MADE IN PROGRAM WRITING; PROGRAM CHECKING

A program, whether in ALGOL or any other programming language, can be a fairly intricate composition, and it is recommended that a systematic check be made for errors and slips before first running it. The computer does not ignore slips; an apparently trifling error in punctuation may well make a program unacceptable, or what is worse, lead to incorrect results. The following list of checks will indicate the type of mistakes most commonly made.

1. General Program Checks

(1) Check that the multiplication symbol * has not been omitted—a common and simple mistake to make and to overlook.

(2) Make sure that no division by zero, no determination of the logarithm of zero or of a negative quantity, no determination of the square root of a negative quantity, and no evaluation of $a \uparrow b$ ($a < 0$, b non-integral) or any other undefined operation can occur.

(3) Check that all numbers are written in the accepted forms.

(4) Check that two arithmetic operators do not appear next to one another.

(5) Check that when testing the magnitudes of quantities the absolute values of these quantities are used.

(6) Check that cycles are nested correctly and that each cycle terminates.

2. Special Checks for ALGOL

(7) Check throughout the program that all basic words used (**begin**, **end**, **for**, etc.) are underlined.

(8) Check throughout for the correct spelling of basic words.

(9) Check for correct punctuation; in particular, check for the correct placing of semicolons. Remember that the first symbol following any statement must either be

; or **else** or **end**

and that **end** must be followed by a semicolon, **else** or another **end** (intervening words are treated as comment). Failure to put a semicolon after **end** will result in the next statement being treated as comment and not translated and consequently not executed. Check also that the word **comment** occurs only after a semicolon or **begin**, and that the comment material is terminated by a semicolon.

(10) Check that for every **if** there is a corresponding **then**. Check also that an **if** never follows a **then**. Remember that a conditional expression must always have an **else** part, although a conditional statement need not. Thus the statement

if A < 0 **then** A := −A;

is equivalent to

A := **if** A < 0 **then** −A **else** A;

and it is incorrect to write

A := **if** A < 0 **then** −A;

(11) Check that all the variables on the left-hand side of an assignment statement are of the same declared type (e.g. a := b := c * d is incorrect if a is **real** and b is **integer**).

(12) Check the correct use of brackets; in particular, check that there are corresponding opening and closing brackets in arithmetic expressions, that the argument for a standard function is enclosed in brackets (e.g. sin (x) not sin x) and that square brackets are used in arrays and around suffixes. Remember the natural order of precedence of the operators in arithmetic or Boolean expressions, and check that brackets have been used to change the order of execution of operations where required.

(13) Check that each **begin** has a corresponding **end**.

(14) Check that each compound statement (i.e. a sequence of statements to be executed together as one unit) is bracketed between **begin** and **end**. This check is particularly necessary in the case of a compound statement following a **for** clause or a compound statement constituting a branch of a conditional statement.

(15) Check that all variables are declared and are not used outside the block in whose head they are declared, and that variables are not introduced into expressions before values have been assigned to them. Check also that identifiers differ in their first six symbols. Check that all labels are declared in a switch declaration at the head of the innermost block in which the statement to which they are attached occurs. Check that declarations occur only at the head of a block, and that the procedure declarations, if any, come last.

(16) Check that the operator $\div$ (or its equivalent form **div**) is used only when both operands are of type **integer**. In particular, note that the result of exponentiation $i \uparrow j$ is **real** even when i and j are of type **integer**, so that $(i \uparrow j \div 2)$ is incorrect.

(17) Check the correct use of labels in **go to** statements. No **go to** statement may lead from outside into a block or from outside a **for** statement to a statement within the **for** statement. No **go to** statement may lead out of the body of a procedure, except through the medium of a label parameter.

(18) For every conditional statement or expression establish two tests, one which makes the arithmetic comparison true and one which makes it false. By following the action of the program for both these cases check that the program always behaves correctly.

(19) When using a **for** clause, such as **for** V:= L **do**, remember that when the list of values of V is exhausted, the variable V cannot be used again in the subsequent program until it has been assigned a value. V retains its current value only when a **go to** statement brings about a jump out of the **for** statement before the list of values of V is exhausted.

(20) Numbers of type **real** cannot in general be held absolutely accurately in the computer. The error is only of the order of two parts in 10^{-9}, but it must be borne in mind when the equality of two **real** numbers is tested for. Thus the relation $A = B$ may have the value **false** at a certain point in a calculation even though A and B are theoretically equal. To avoid errors of this sort, write the relation instead as $\text{abs}(A - B) < \text{epsilon}$, where epsilon is a suitably small positive constant. This inaccuracy in the computer representation must also be taken into account in **for** statements. Thus

for A := 1 **step** 1/3 **until** 2 **do** S;

could result in the statement S being executed for $A = 1, 1\frac{1}{3}, 1\frac{2}{3}$, but not for $A = 2$. On the other hand,

for A := 1 **step** 1/3 **until** 2 + epsilon **do** S;

is safe.

The list given here is not exhaustive.

It is good practice to work through the program with a set of test data, to make sure that it behaves correctly at every stage. This is often the best way of locating errors.

APPENDIX 3

ALGOL 1 EXTENSIONS

CONTENTS

1. INTRODUCTION

For each of the new facilities listed in section 2 and described in more detail in section 4, a modification tape is available.

An ALGOL system incorporating one or more of the new facilities may be produced by using these tapes, and tapes 1 and 2 of ALGOL 1 (Issue 1).

The MODS must not be used with ALGOL 1 (Issue 2).

2. SPECIFICATION OF FACILITIES

2.1 Arrays on Core Backing Store (MOD 2)

All array elements are held on a dynamic stack on core backing store . No special text is required; array declaration and accessing are written as for ALGOL 1 (Issue 1).

2.2 Lineprinter facility (MOD 3)

The facilities made available by this modification are based on those of LPRALG (see MANUAL section 2.2.3. 24).

The procedure calls added by MOD 3 to ALGOL 1 (ISSUE 1) are:

lineprinter

top of form

punch (4)

The LPRALG procedures overwrite, lines (i), find (i) are not available.

2.3 Editing facility .

The object text supplied to the compiler may be modified by a suitable edit tape in reader 2. This tape is written in the same manner as for EDIT 8.

2.4 Segmentation — using Core Backing Store (MOD 8)

This is to allow programs normally too large for ALGOL 1 (ISSUE 1), to compile and run by using core backing store. The program may be divided into segments which are held on core backing store and brought to main store only when required.

2.5 Segmentation - using Magnetic Tape (MOD 9)

This modification provides the same facility as MOD 8 but uses Magnetic Tape instead of Core Backing Store.

2.6 ALGOL/STAR (MOD 10)

This modification makes possible the running of ALGOL programs within the STAR system. (see section 2.5.5 of the MANUAL).

2.7 Improvement of machine code facilities (MOD 11)

This modification gives a machine code facility based on that in Elliott Autocode. It dispenses with repeated use of the word elliott (as in "elliott" orders) and use of commas as separators. Relative addressing with respect to the start of a unit of code is allowed.

The restrictions on the use of identifiers in machine code are as in elliott. Elliott instructions as presently written will, however, still be accepted.

2.8 Standard procedure improvements (MOD 12)

This modification enables boolean expressions to stand in print statements and introduces two procedures, one affecting character handling, the other causing a return to the presumed settings for read and print statements. It also modifies the continuation values used after certain errors have been detected in the standard procedures "sqrt", "exp" and "ln".

3. METHOD OF USE OF THE MODS

Each MOD should be input by typing IN. after ALGOL 1 (ISSUE 1) systems tapes have been input (or are already in store). Operating instructions are as follows:

Step	Tapes	Instruction to machine	Indication on machine
1	RAP or RAPMT	depress no protection	
		press reset button	
		press clear store	
		press initial instructions	RAP or RAPMT
2	ALGOL 1 tape 1	release no protection	
		press reset button	
		press message	?
		IN.	ALGOL 1
			Swait
3	ALGOL 1 tape 2	change key 39	ALGOL 2
			Swait

Step	Tapes	Instruction to machine	Indication on machine
4	MOD tape(s)	press message button IN.	? ALGOL 1 UNCHEK- ALGOL 2 UNCHEK
5	Object program	ALGOL.	as usual.

If two or more MOD tapes are to be input this must be done in the order given in the table below. MODS with numbers to the left must be input before MODS with numbers to the right.

MOD ORDER	5	10	3	12	8	9	2

MOD11 can not be used with any other MOD.
MOD9 can not be used with MOD2.
MOD12 can only be used with MOD3 and/or MOD5.
MOD8 can not be used with MOD9.

4. DESCRIPTIONS OF MODS

4.1 ALGOL 1 MOD 2 - Arrays on core backing store

FUNCTION

MOD 2 changes ALGOL 1 so that all arrays declared by the programmer are held on core backing store. No special form of text is required for declaration or accessing of arrays. Subscript bounds are checked as in ALGOL 1. The optimising standard functions lowbound (A,I), range (A,I) and size (A) have the same values as in ALGOL 1. The integer procedure address (A) gives the core backing store address of the first element of the array. The variable location I has the (real) value of the contents of backing store location I.

STORE USED

The total mainstore space occupied by MOD 3, MOD 8 and MOD 2 together is:

compile time	501 locations
run time	249 locations

Because of the way MODS are coded, no space may be saved, if MOD 2 is used, by not using MOD 3 or MOD 8. At run time, MOD 2 places the own arrays at the high addressed end of core backing store, and the others at the low addressed end. If MOD 8 is used, the program segments are at the low addressed end and the non-own arrays follow them. MOD 8 and MOD 2 automatically determine the size of core backing store, and use all of it.

TIME

Access time for array elements held on core backing store is approximately 25% greater than for mainstore. The time increase for a given program will, however, vary with the nature of the program.

ERROR INDICATIONS

The run-time error indications of ALGOL 1 are extended as follows:-

Subscript oflo

When a subscript is out of range, the following are displayed, each on a new line, on the typewriter.

(a) the subscript position at which the error occurred.

(b) the value of the erroneous subscript.

(c) the current upper and lower bounds of each subscript position of the array concerned, subscript position 1 first.

(d) the message

```
                    Subscript oflo
                    Swait
```

The print format is digits (6). If the erroneous subscript is very large ($\leqslant$1 000 000) it will as usual, appear in the format scaled (2).

Space oflo

When an array declaration or value array copy finds that there is not enough space in core backing store, the message

```
                    CBS   Space oflo
                    Swait
```

is displayed on the typewriter.

OTHER MODS

For use with other MODS see section 3.

TAPE

The tape supplied is in RAP binary code.

METHOD USED

All array elements are held on a dynamic stack on core backing store. A system of pointers is held as a tree structure on the dynamic stack in main store. Accessing of elements is done by a single word transfers.

KNOWN ERRORS IN ALGOL 1 MOD 2 ISSUE 1

(i) In the subscript overflow diagnostics, the value given for the subscript position at which the error occurred is 1 less than the true subscript position if the value of the erroneous subscript is less than the lower bound of its subscript position.

(ii) The standard procedure "location" should not be used.

(iii) If a running program is interrupted and

REPEAT.

is typed, then any space which had been claimed on backing store for arrays by the program will not be given back to the dynamic routines.

If however, the program is allowed to run to its conclusion before

REPEAT.

is typed, then all the array space will be made available to the dynamic routines.

(iv) The control variable in a "for" statement should not be a name parameter or a subscripted variable.

(v) The first term in an arithmetic expression on the RHS of an assignment statement should not be a name parameter when the corresponding actual parameter is a subscripted variable.

e.g.

```
begin integer a ;
integer array A[1 : 10];
procedure F(x) ; integer x ;
      begin
      a : = x div 3 ;
      print a ;
      end ;
   :
F(A[10]) ;
end ;
```

This would give an incorrect result. However, if the assignment statement in the body of the procedure were changed to

```
a : = 1 * (x div 3) ;
```

then the correct result would be obtained.

(vi) When an arithmetic expression is used for a name parameter in a procedure call, the expression should not begin with a subscripted variable or a name parameter.

```
e.g.    begin integer array A[1 : 10];
              procedure F(x); integer x ;
              begin
              print (x) ;
                .
                .
                .
              end ;
        .
        .
        .
        F(A[10]+1) ;
        end ;
```

This would give an incorrect result. If, however, the call of the procedure were changed to

```
        F(1 + A[10]),
```

then the correct result would be obtained.

(vii) If an array bound is a constant greater than 8191 (or less than - 8191) then the bound is not interpreted correctly.

(viii) The error message "Int oflo" is displayed instead of 'CBS space oflo'.

4.2 ALGOL 1 MOD 3 - Lineprinter Facility

FUNCTION

Provides two additional procedure calls (i.e. those made available in LPRALG):-

(a) lineprinter

(b) top of form

STORE USED

MOD 3 extends the dynamic routines by 61 locations.

ADDITIONAL PROCEDURES

Lineprinter

A call of lineprinter has the same scope as other device setting procedures. Thus its effect within a print statement is local to that statement, otherwise global.

Lineprinter causes characters output by print statements & oustring, to appear on the lineprinter. The first such character appears at the beginning of a line. A line is printed only when, on paper tape, a newline character would be output, or when there is no room in the buffer, for the character sent. In this case, the character appears at the beginning of the next line.

All lower case characters appear in upper case, and other characters not in the lineprinter character set appear as the triangle character (Δ). An underlined character appears as an underline followed by the character i.e. as two separate characters. Similarly for vertical bar.

<u>top of form</u>

This has the effect that 'top of form' is done before printing the next line, i.e. paper is fed until the next top of form character on the control loop is found. If there is no control loop in the lineprinter, top of form has no effect.

PROCEDURES NOT AVAILABLE

The LPRALG procedures overwrite, lines (i), find (i) are not available.

The effect of the procedure lines (i) can be obtained by declaring at the beginning of a program

```
procedure  lines(i);  value i;  integer i;
begin  integer i;
end;   for i:= 1 step 1 until i do print lineprinter, ££l??;
```

The procedure call

punch(i);   when i=4,

has precisely the effect of

lineprinter;

ERROR INDICATIONS

LPerr	is displayed when an error condition arises in the line-printer when it is in use . (lineprinter in manual, paper low etc.). When the error condition is rectified the program continues automatically.

ERRINT4 is displayed if an attempt is made to use a lineprinter which is disconnected or switched off.

NOTE

1. For use with other mods see section 3.
2. The first call of lineprinter (or punch (4)) sets up a 121 word buffer at the low addressed end of the free store area, overwriting part of the compiler. It remains there for the rest of the program and may not be overwritten by dynamic arrays. As a consequence, the system must be reinput after the lineprinter has been used, before the next program may be translated.

TAPE

The tape supplied is in RAP binary code

4.3 ALGOL 1 MOD 5 - EDITING FACILITY

FUNCTION

This modification enables the compiler to use the library program EDIT 8 as a common program. Source programs may then be edited directly into store.

STORE USED

MOD 5 extends the dynamic routines by 27 locations.

METHOD OF USE

EDIT 8, the systems tapes and the MOD 5 tape are input to store. The edit commands to modify the program text are punched on paper tape and loaded in reader 2. The tape to be edited is loaded in reader 1.

The entry to ALGOL to read characters from reader1 only is

ALGOL.

Entry for editing into store is

ALGOL; EDIT8.

The edit tape and the tape to be edited are read by EDIT 8 and the edited characters are presented one at a time to the ALGOL compiler. An edited tape is not produced.

Alternative Common Programs.

Any binary common program symbol read routine written in SAP 1, having the characteristics listed below, can be used in place of EDIT 8. The entry to ALGOL for symbol read by common program N is

ALGOL ; 'N'.

Initial entry from the ALGOL compiler to the symbol read routine is at A + 1, where A is the address of the first trigger point. This is to initialise workspace of the routine. Thereafter, at entry point A + 2, a single character of the source program is brought into the accumulator, the overflow register remaining undisturbed. In each case, exit is by EXITCP instruction.

Thus, part of the symbol read routine has the form:

1st trigger point	40 ordinary entry:
common program entry 1	40 initialise w/s:
common program entry 2	40 get character :

NOTES

(i) The address of the first trigger point of the symbol read routine is used only to generate the addresses of the common program entries.

(ii) If a source program is compiled using the symbol read facility and then successfully run so that

Swait

is output, on changing key 39 the subsequent compilation will again use the symbol read facility.
Should the run be unsuccessful then press the message button and type either

ALGOL ; 'N' .

where N is the symbol read routine, to restart compilation, or

ALGOL.

to compile a fresh program which requires no modification.

(iii) ALGOL programs small enough to fit in main store, but large enough to overwrite Algol tape 1, can be compiled and run and the message Swait output.
The message

Reinput tapes 1 and 2

is output, when key 39 is changed.

(iv) The editing facility can be used when outputting owncode to paper tape. The entry to edit and assemble is

OWNOUT ; EDIT 8.

The normal entry remains

OWNOUT.

Other common program symbol read routines having the form given above can also be used.

2.1.3

OPERATING INSTRUCTIONS

STEP	TAPES	INSTRUCTION TO MACHINE	INDICATION ON MACHINE
1.	RAP or RAPMT	depress no protection	
		press reset button	
		press clear store	
		press initial instructions	RAP or RAPMT
2.	Symbol read Routine 'N'	release no protection	
		press reset button	
		press message button	?
		IN.	'N'
3.	ALGOL 1 tape 1	IN.	ALGOL 1 Swait
4.	ALGOL 1 tape 2	change key 39	ALGOL 2 Swait
5.	MOD 5 (edit)	press message button	?
		IN.	ALGOL 1 UNCHEK

6. <u>Either</u>

Source program in reader 1 and edit tape in reader 2:	ALGOL ; N.
<u>Or</u>	
Source program in reader 1 :	ALGOL.

<u>OTHER MODS</u>

For use with other MODS see section 3.

<u>TAPE</u>

The tape supplied is in RAP binary code.

4.4 ALGOL 1 MOD 8 - Segmentation facility

FUNCTION.

This modification allows the user to run programs too large for the available main core store by making use of core backing store.

The program may be divided into segments (these can be numbered) which are held on core backing store and only brought to main store when obeyed. The programmer has complete control over the way in which the program is segmented (see METHOD OF USE).

'Owncode' produced by the compiler is automatically written to core backing store.

STORE USED

MOD 8 extends the dynamic routines by 121 locations, and the compiler by 167 locations.

METHOD OF USE

The programmer may specify that a block is a segment by placing a significant comment immediately after the begin of the block concerned. In addition, a segment area number may be specified. It is not necessary to specify consecutive numbers for segment areas. The areas may be numbered from 1 to 16 inclusive; If a particular number is not used, that area has zero size.

```
begin  comment  segment [6] non-significant comment;
 real  a, b ;
    etc.
     .
     .
     .
     .
```

The part of the comment which follows the character close square bracket is not significant and may be used for the ordinary purposes of comment.

A block which is specified as a segment may not contain another block specified as a segment; ie. segments may not be nested. (See ERROR INDICATIONS for further information).

2.1.3

At compile time, for each segment area number, an area of the main store, which is known as the segment area, is set aside; this area is big enough to contain the largest segment with that number. The main program (i.e. all the text not contained in the segments) remains in main core store throughout; only one segment may be in main store at one time, and it must be held in the segment area. A copy of the code of each segment is kept on core backing store, always in the same position. No modification of instructions is required when the segments are brought to main store and, thus advantage is taken of the ADT 'tag' protection on the peripheral transfer inasmuch as the segment may be obeyed as soon as the first words of the segment are transferred to main store.

When a segment block is entered, if it is not already in main store, it is copied into the segment area corresponding to its 'segment area number'. Thus segments with the same number may not be in main store together.

If a segment is brought to main store and re-entered several times before another segment is entered it is not re-copied from core backing store each time.

The programmer may include in a segment a call of a procedure whose body is also a segment. In general, provided the rules for the inclusion of the significant comments are obeyed, no programming restrictions are imposed.

If segment area numbering is not used

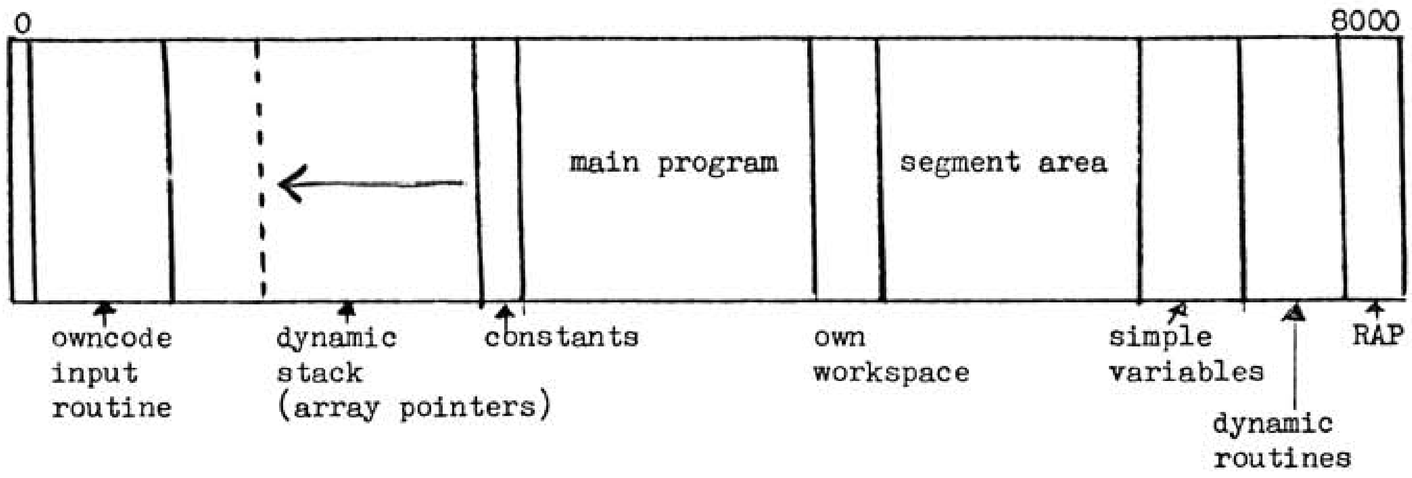
e.g.

```
                begin  comment   segment :;
                   real   ab;   etc.
```

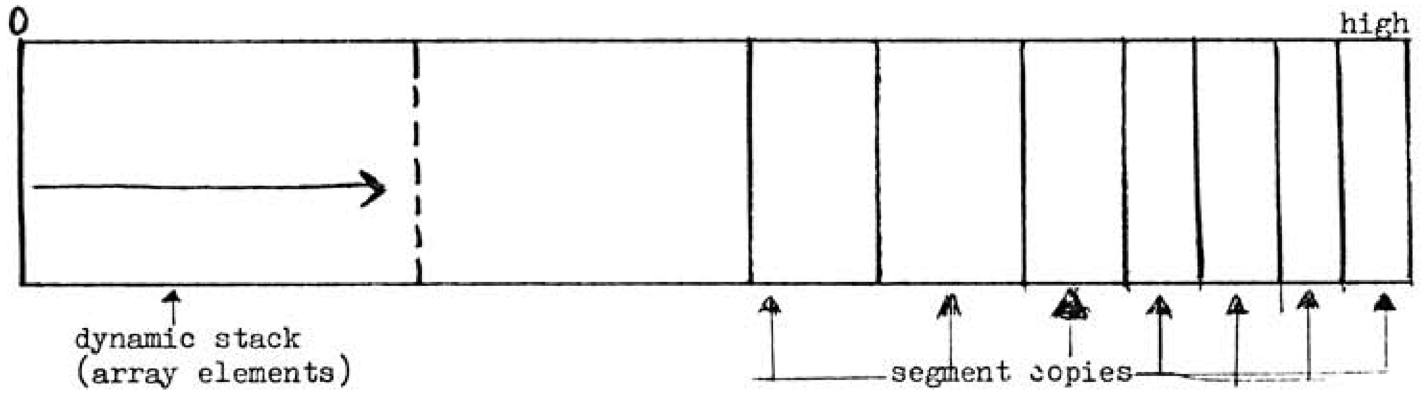
this is taken to mean

```
                begin  comment   segment [1];
                   real ab; etc.
```

After compilation, the main store picture is:



and core backing store picture is:



ERROR INDICATIONS

To new compile-time indications are introduced. The details are as follows:

Indication.	Interpretation
Error No. 55	(a) Nesting of segments has been attempted.
	(b) The significant comment has been placed illegally, e.g. an attempt has been made to segment a compound statement.

Indication	Interpetation
BSOFLO	The program is too large to be compiled. This indicates that the space taken by the owncode at compile time is greater than that available or that, when the owncode is converted to program, the program segments thus created in backing store plus the remaining owncode fill the space available. In this case the solution is to use MOD 9 provided that magnetic tape is available.

TAPE

The tape supplied is in RAP binary code

SPECIAL FEATURE

To assist the programmer in the decisions taken as to where the program may best be segmented a facility is provided whereby the size of each segment created is displayed on the typewriter at compile-time. This facility is invoked by depressing key 38 on the number generator. As each segment is completed the size of each segment is printed as a 4-digit integer with no suppression of leading zeros.

OTHER MODS

For use with other mods see section 3.

4.5 ALGOL 1 MOD 9 - Segmentation Using Magnetic Tape

FUNCTION

This modification allows the user to run programs too large for the available main core store by making use of magnetic tapes (handlers 4 and 5).

The program may be divided into segments which are held on magnetic tape and only brought to main store when obeyed. The programmer has complete control over which blocks in his program are segmented.

Very large programs may be run using this modification. In the case where a program is too large for a machine with core backing store, or because the core backing store is required for arrays, this modification may be used. All segments will be held on magnetic tape. No facility will be provided on a machine with both core backing store and magnetic tapes for placing some segments on magnetic tape and some on core backing store.

'Owncode' produced by the compiler is automatically written to handler 4.

STORE USED

MOD 9 adds 174 locations to the dynamic routines and 167 locations to the compiler.

METHOD OF USE

The programmer may specify that a block is a segment by placing a significant comment immediately after the begin of the block concerned. In order that the tape movement may be minimised, each segment may be categorised so that segments which make repeated calls on each other, or which are repeatedly called by the main program, may be in main store at the same time. Thus,

```
begin  comment  segment [7] non significant comment;
     real a, b ;
        etc.
         .
         .
         .
         .
```

specifies that the segment will occupy segment area number 7. It is not necessary to specify consecutive numbers for segment areas. The areas may be numbered from 1 to 16 inclusive. If a particular number is not used, that area has zero size.

An alternative style of segment specification is accepted:

```
begin  comment   segment : non significant comment ;
     real a, b ;
        etc.
         .
         .
         .
         .
```

This is taken to specify segment area 1.

In all cases, the part of the comment which follows the colon or close square bracket is not significant and may be used for the ordinary purposes of comment.

A block which is specified as a segment may not contain another block specified as a segment; i.e. segments may not be nested. (see ERROR INDICATIONS for further information).

The programmer may include in a segment a call of a procedure whose body is also a segment. In general, provided the rules for the inclusion of the significant comments are obeyed, no programming restrictions are imposed.

ERROR INDICATIONS

A new compile-time indication is introduced. The details are as follows:

Indication	Interpretation
Error no. 55	Either (a) Nesting of segments has been attempted, Or (b) The significant comment has been placed illegally, e.g. an attempt has been made to segment a compound statement.

N.B. The user is recommended to add this detail to the list of error numbers on page 34 (2.1.3).

The following error indications may occur either at compile or run time, when reading from magnetic tape:

ERR X	Systems error which concerns the numbering of magnetic tape blocks.
ERR LS	A required block is found to be long or short.
ERR P	After 64 re-reads, a required block still gives a parity error.

In each of the above situations no continuation is possible.

SPECIAL FEATURE

To assistthe programmer in the decisions taken as to where the program may best be segmented a facility is provided whereby the size of each segment created is displayed on the typewriter at compile time. This facility is invoked by depressing key 38 on the number generator. As each segment is completed the size of the segment and its segment area number are displayed as two 4-digit integers with no suppression of leading zeros.

METHOD USED

At compile time, for each segment area number used, a segment area is allocated which is large enough to hold the largest segment with that number. The main program (i.e. all the text not contained in the segments) remains in main core store throughout.

A copy of the code of each segment is held on magnetic tape and is automatically brought to main store when the segment is entered. If that segment is then re-entered several times before another segment of the same numerical category is entered then it is not re-copied from magnetic tape each time.

TAPE

The tape is coded in the normal RAP binary form.

OTHER MODS

For use with other MODS see section 3.

4.6 ALGOL 1 MOD 11 - Improvement of Machine Code Facilities

This modification is included in the ALGOL 3 compiler and the facility is as described in section 2.1.5.4.7.1 of the MANUAL.

4.7 ALGOL 1 MOD 12 - Standard Procedures Modification

FUNCTION

(a) To permit boolean expressions to stand in print statements.

(b) To modify the continuation values used after errors arising in the procedures, "sqrt" "exp" and "ln," and to enable the run of an ALGOL program to be continued after the generation of a floating point overflow. Two new standard procedures, "noflo" and "oflo" have been introduced in connection with these changes.

(c) To introduce the format setting procedure "presume".

(d) To introduce the character handling standard procedure "character".

2.1.3

GENERAL DESCRIPTION

(a) BOOLEAN EXPRESSIONS IN PRINT STATEMENTS

Boolean variables, procedures and expressions may stand in print statements. The string 'true' or 'false' will be output to the current device.

(b) CONTINUATION VALUES

The following changes have been made to the continuation values, after an error has been detected in the standard procedures "ln", "sqrt", and "exp". The new error message "log zero" has been introduced. In the description below, x stands for argument of the function which caused the error and cv the value which the function will assume if the program is continued.

Message	Meaning
exp error	$x\ 255 \log_e 2$ ($\triangleq 176.75$) $cv = (1 - 2^{-29}) \times 2^{255}$ (i.e. largest positive number the computer can hold).
log zero	$x = 0$ $cv = -2^{255}$ (largest negative number the computer can hold).
log error	This is caused by:- (i) ln (x), where $x < 0$. The continuation value is $\ln(\lvert x \rvert)$ (ii) attempted evaluation of $p \uparrow q$ with $p \leqslant 0$ and q real and $\leqslant 0$. The continuation value of the function is zero.
sqrt error	$x < 0$. $cv = \text{sqrt}\ (\lvert x \rvert)$.

(i) Suppression of certain error messages

Two standard procedures, "noflo" and "oflo" have been introduced, which can be used to suppress some of the error messages and modify the values given to certain of the standard functions with particular arguments. Used in conjunction with the SAC program ERINT (FOR MOD 12), "noflo" can also be used to suppress the error message ERRINT 1.

(ii) noflo

This is a procedure without parameters which, when called, affects the standard functions "ln", "sqrt", and "exp" in the manner described below.

(i) ln(0) The message "log zero" and the "Dwait" are not displayed. The continuation value is as before, i.e. -2^{255}.

(ii) exp(x) where $x > 255 \log_e 2$ ($\doteq 176.73$)

The message "exp error" and the "Dwait" are not displayed. The continuation value is as before, i.e. $(1-2^{-29}) \times 2^{255}$.

(iii) sqrt (x) where $x = (1-2^{-29}) \times 2^{255}$

This expression takes the value $(1-2^{-29}) \times 2^{255}$. Since sqrt (x) when $x<0$ is evaluated as sqrt ($|x|$) then, when $x = -(1-2^{-29}) \times 2^{255}$, this expression will also be given the value $(1-2^{-29}) \times 2^{255}$. The error message "sqrt error" will still be displayed, however, when x is negative.

(iv) ln (x) where $x = (1-2^{-29}) \times 2^{255}$

This expression takes the value $(1-2^{-29}) \times 2^{255}$. Since ln(x) where $x<0$ is evaluated as ln ($|x|$) then, when $x = -(1-2^{-29}) \times 2^{255}$ this expression will also be given the value $(1-2^{-29}) \times 2^{255}$. The error message "log error" will still be displayed, however, when x is negative.

(iii) oflo

This is a procedure without parameters and is used to cancel the procedure "noflo". The action taken in the cases listed under (ii) noflo then reverts to that described above under CONTINUATION VALUES.

The procedure which is called last in the dynamic sense is the one that is operative. The presumed state corresponds to a call of "oflo".

(iv) <u>Continuation after floating point overflow, i.e. ERRINT 1.</u>

A SAC program ERINT (FOR MOD 12) - described in section 5.2 - has been provided to enable the run of an ALGOL program to be continued after the detection of a floating point overflow.

If ERINT (FOR MOD 12) is in store when the error interrupt occurs, RAP or RAPMT will enter ERINT (FOR MOD 12) which, unless the procedure "noflo" is currently active in the ALGOL program , will display

"ERINT 1
Dwait"

When the leftmost F2 digit (key 19) of the word generator is changed, the ALGOL program continues with the value $(1-2^{-29})\text{x } 2^{255}$ taken as the result of the operation which generated the floating point overflow.

If the procedure "noflo" is active in the ALGOL program then the run will be continued without an error message and with the continuation value $(1-2^{-29})\text{x } 2^{255}$, (see ERINT (FOR MOD 12) description, section 5.2)

(c) PRESUME

A new standard procedure "presume" has been introduced. It has no parameters and may be used to restore all the setting procedures to their presumed values.

```
i.e. reader(1)
     punch (1)
     digits(8)
     freepoint(8)
     prefix (££l??)
     leadzero (£?)
```

"grouping (n)" and "special (n)" are cancelled.

It may be used globally (i.e. affecting all <u>read</u> and <u>print</u> statements) when placed outside a <u>read</u> or <u>print</u> statement.

When placed inside a <u>read</u> statement the procedures affecting input are changed locally (i.e. only within the <u>read</u> statement) and procedures affecting output are changed globally.

When placed inside a <u>print</u> statement the procedures affecting output are changed locally and those affecting input are changed globally.

(d) CHARACTER

A new standard integer procedure "character (n)" has been introduced. This procedure is used in conjunction with the standard procedure "advance (n)". "character (n)" takes the 503 paper tape code value of the character currently in the buffer for input device n. No character is actually input when "character (n)" is obeyed.

If n does not satisfy

$$1 \leqslant n \leqslant 3$$

then "character" takes the value of the character currently in the buffer for reader 1.

<u>TAPE</u>

A tape coded in RAP binary form is supplied.

<u>STORE USED</u>

MOD 12 adds 58 locations to the compiler and 109 locations to the dynamic routines.

<u>OTHER MODS</u>

For use with other MODS see section 3 of this appendix.

5. SAC PROGRAMS

There are two programs that may be required when using the MOD tapes, INSERT and ERINT (FOR MOD 12). They are described below.

5.1 INSERT

<u>CODE</u>

INSERT S

<u>FUNCTION</u>

To enable the user to make new ALGOL systems tapes which include the modifications contained on one or more relocatable binary MOD tapes thus simplifying operating, especially for installations with no magnetic tape.

<u>STORE USED</u>

Program 234 locations - workspace 24 locations.

METHOD OF USE

A MOD tape contains two separate modifications, one to tape 1 of the compiler and one to tape 2. The modification to tape 1 ends in a trigger to read in the modification to tape 2, and a short section of blank tape separates the two. As INSERT places the modification to tape 1 in tape 1 and the modification to tape 2 in tape 2, the MOD tape, for convenience should be cut at this point.

The MOD tapes must be inserted in a particular sequence. A table is provided below which specifies the sequence where necessary.

Operating Instructions.

Stage 1 Load INSERT tape (binary version) in reader 1 and type IN.

Stage 2 Load ALGOL tape 1 or ALGOL tape 2 as appropriate in Reader 1 and the first MOD to be inserted in Reader 2.

Stage 3 For ALGOL tape 1:-

Action	Result
type INSERT.	ALGOL tape 1 will be read in almost to the end, followed by the first MOD to tape 1 in Reader 2. The new tape will be produced on punch 1.

For ALGOL tape 2:-

	Action	Result
(i)	Type INSERT;2.	The message LAST MOD NO = will be displayed.
(ii)	Type the number of the final MOD to be inserted (see table below) and terminate with a full stop.	ALGOL tape 2 will be read in almost to the end, followed by the first MOD to tape 2 in Reader 2. The new tape will be produced on punch 1.

When the necessary code has been copied from the MOD, the message

MORE MODS ?

will be displayed.

Stage 4	Load the next MOD in Reader 2 and type YES.	When the MOD has been copied, the message MORE MODS ? will be displayed. Repeat stage 4 for each MOD.

When there are no more mods to be inserted:-

Stage 5	Type NO.	The final MOD must be left in reader 2. The rest of the systems tape in Reader 1 will be read and the program will come to a loop.

To check the output tape for punching errors, place the leading blanks in Reader 1 and change the sign key (key39). When the tape has been input the message

Tape Correct

or Incorrect

will be displayed as appropriate. If the 'Incorrect' message is displayed, to cater for the possibility of a mis-read on checking, the checking process may be repeated by again loading the tape in Reader 1 and typing:-

INSERT;3.

ERROR MESSAGES

NOT 80	This will be displayed if the first non-blank character of a tape in Reader 1 or Reader 2 does not have the value 80. (All RAP binary tapes begin with this character). Rectify the fault and continue by changing the keyboard sign key (Key 39).
WRONG LAST MOD NO	The number used to answer the question LAST MOD NO = was not acceptable. To continue, change the keyboard sign key (Key 39). The question will be asked again.

ORDER OF MODS

This table gives the order in which MODS must be inserted into the systems tapes. MODS with numbers to the left must be inserted before MODS with numbers to the right.

Tape 1

3 5 10 12 8 9 2

Tape 2

5 10 3 12 8 9 2

Do not insert MODS into systems tapes which contain previously inserted MODS. Instead insert all the required mods into the original systems tapes.

Certain of the mods are not compatible and so may not be inserted together. See section 3 of the appendix for the list of restrictions.

Note MOD 11 may not be used with INSERT.

TAPE The tape is coded in SAC and it is recommended that a binary tape is produced, entering SAP with keys 35 and 36 of the word generator depressed.

5.2 ERINT (FOR MOD 12)

CODE

ERINT S

FUNCTION

This program is used in conjunction with the procedures "noflo" and "oflo" available in ALGOL 1 MOD 12 (see section 4.7). It enables the run of an ALGOL program to be continued when a floating-point overflow is generated. If the procedure "noflo" is currently active in the ALGOL program, no message will be displayed. If the procedure "oflo" is called, the message

ERINT 1
Dwait

will be displayed, after which, the program may be continued. In both cases the value $(1-2^{-29})\times 2^{255}$ is taken as the result of the operation which generated the error interrupt.

STORE USED

71 locations, including workspace.

METHOD OF USE

ERINT (FOR MOD 12) is written as a SAC common program. It should be placed in store before ALGOL 1 (ISSUE 1).

There are two entry points:-

Entry point 1

This is an initialisation entry used by the dynamic routines. It modifies RAP or RAPMT and the program head of ERINT (see PROCESS USED). Exit is made by the order

EXITCP,2.

Entry point 2

This is the point at which RAP or RAPMT enters ERINT when an error interrupt is generated. Unless the interrupt has been caused by a floating-point overflow ERINT will return control to RAP. If there has been a floating-point overflow; ERINT will cause the program to continue with or without an error message (see PROCESS USED).

PROCESS USED

Entry point 1

The Dynamic Routines will search main store for the program ERINT as soon as the run of an ALGOL program is started, and if it finds,it, the Dynamic Routines will enter ERINT at the 1st location after the LINKCP. Here ERINT modifies RAP so that RAP will enter ERINT as soon as an error interrupt is detected. Since this entry has to be made to the second entry point of ERINT, the main entry instruction in the program head of ERINT is also modified so that it contains a jump to the second entry point.

Entry point 2

When an error interrupt is detected, RAP will enter ERINT at the main entry point (in the program head) which will have been modified to cause a jump to the second entry point of ERINT. Here the error interrupt locations are examined (see Sections 1.2.3, 1.2.4 and 1.2.5 of the Manual) to ensure that the error interrupt was caused by a floating-point overflow.

If this condition is not satisfied, ERINT returns control to RAP which will display the error message (for the format of the error message see Note below). However, if a floating-point overflow has occurred, ERINT will examine a marker in the dynamic routines which is set when the procedure "noflo" is active. If this marker is not set then the message

ERINT1

Dwait

is displayed.

The ALGOL program may then be continued if the leftmost F2 key (key 19) of the word generator is changed.

If the marker is set, i.e. "noflo" has been called, then no error message will be displayed and the ALGOL program will be continued.

Exit is made from ERINT using the 66 instruction(see Section 1.2.4). Upon continuation the overflow register has the value it had when the floating-point overflow was generated and $(1-2^{-29}) x\ 2^{255}$ (i.e. the largest positive number the 503 can hold), is taken as the result of the operation that caused the error.

TAPE

A mnemonic tape coded in SAC is supplied. For use with ALGOL 1, (ISSUE 1) a binary tape must be produced using SAP1. The program is sum-checkable.

Note Once entry point 1 of ERINT has been used, RAP will remain in the modified form, unless it is re-input from paper tape. However, unless a program called ERINT is in store, RAP or RAPMT may be used in the normal manner, except that should an error interrupt be generated, the format of the message displayed will be:

ERINTm

(where m is the number of the error interrupt) instead of

ERRINTm

after which RAP will come to a typewriter demand.

The symbol ">" (see 2.2.1.4 of the 503 Manual) will not be displayed by this modified version of RAP or RAPMT when entry to a program has been made.

APPENDIX 4

ALGOL 1 ISSUE 2

1. Known Errors in ALGOL 1 ISSUE 1 Removed in ISSUE 2

1.1 Arithmetic Expressions

(i) String, label and switch identifiers in Arithmetic Expressions are not rejected.

(ii) Unary "+" is not accepted.

(iii) Assignment of a Boolean type procedure to an Arithmetic variable is not rejected.

1.2 Arrays

(i) Integer Array bounds greater in modulus than 8191 are incorrectly compiled.

(ii) An integer procedure used as an array subscript on RHS of an assignment statement is incorrectly compiled.

1.3 Numbers

(i) During compilation a number of the form "$a_{10}b$" where "a" has "b" digits after the decimal point gives Error No. 1.

(ii) Upper case 10 and 11 are accepted.

(iii) Integer conversion of a real number greater than 2^{37} but less than $(2^{39}-1)$ leads to "INT OFLO".

1.4 Identifiers

Identifiers beginning with A, B or C are differentiated by the first 7 characters instead of the first 6.

1.5 "for" Statement

Control variables in "for" statements which are not identifiers or array elements of type real or integer are not rejected.

1.6 Error Messages

(i) An ampersand leads to a non-continuable error.

(ii) Error No. 49 occurs prematurely during FAME checking of some programs.

(iii) In certain circumstances a missing second bracket in an array identifier leads to a non-continuable error with a possible corruption of the system.

1.7 Comment Facilities

If the comment after the word <u>end</u> contains an underlined word immediately before the semi-colon, Error No. 17 is displayed, and if the comment is terminated by an <u>end</u> containing a double underline, this <u>end</u> is ignored.

1.8 Errors When Compiler is Overwritten

(a) During compilation of a program.

This leads to various errors dependent on the amount of the compiler which is overwritten. Some of the more common symptoms displayed are:-

(i) Incorrect Free Store Message.
(ii) No Free Store Message.
(iii) ERRINT 5.

(b) During Running of a Program.

(i) The display of "SPACE OFLO" automatically leads to the display of the Reinput tapes message whether or not the compiler has been overwritten.
(ii) "Reinput tape 2" is displayed instead of "Reinput tapes 1 and 2" and vice-versa.

(c) On typing "ALGOL".

Instead of the reinput tapes message, "INT OFLO" is displayed.

1.9 Standard Procedures

(i) The standard procedure "checkb" is not available, though the equivalent "checkB" is available.
(ii) If "storemax" is used the compiler enters a loop.
(iii) "arcsin (-1)" is given as $\pi/2$ instead of $-\pi/2$
"arccos(1)" is given as π instead of 0.
(iv) "checkB" displays "true" instead of "false" and vice-versa.
(v) If the standard procedure "reader(n) is called and n does not satisfy $1 \leq n \leq 3$, "special(4)" is cancelled.
(vi) In an elliott order a jump to a label in an outer block is not rejected.

1.10 Halt Codes

(i) A halt code which occurs in a string or a title does not cause an "Swait" and is also packed.

(ii) When using "instring" a "Dwait" does not occur if a halt code is found before the leading £ of the string.

2. Restrictions in ALGOL 1 ISSUE 2

(i) Multiple placings of labels will not be detected.

(ii) The following obscure case of recursion will produce incorrect results:-

A procedure A which contains a procedure B which calls A must not have any declarations (other than procedure declarations) preceding a call of B.

```
example   procedure A;
                    • • • •
                    • • • •
                       begin procedure B;
                             begin   ⎤
                             : : : : ⎥
                             A;      ⎥body
                                     ⎥of
                             : : : : ⎥B
                             end;    ⎦
                       begin real p,q,r;                 body
                             : : : :                     of
                             B;                          A
                             : : : :
                       end
                       end;
```

The contents of p,q,r would be lost on entry to B.

Provided B is not a type-procedure, this may be avoided by making all declarations before declaring B.
If B is a type procedure used in a complex arithmetic expression, incorrect results may also be produced even though there are no declarations after the body of B and before its use.

(iii) A procedure may not contain a "for" statement having a recursive call of the procedure within the body of the statement if there is more than one "for" list element.

e.g. <u>procedure</u> wrong(n)

```
    : : : :
    for i:= 1, 2, 3, 4 do wrong (i)
```

In this case, this could be overcome by:-

```
    procedure right(n);
    : : : :
    for i:= 1 step 1 until 4 do right(i)
```

<u>Note</u>

(1) tapes precompiled by ALGOL 1 (issue 1) will not be accepted by ALGOL 1 (issue 2).

(2) the present MOD tapes cannot be used with this issue.

ELLIOTT ALGOL PROGRAM SHEET

Title..........

Subtitle..........

Issue	Sheet	of

Elliott Brothers (London) Ltd., Borehamwood, Herts.

N.B. Suppliers of Elliott Algol Program Sheets are Cresta Press Ltd, Holywell Industrial Estate, Watford, Herts.

503 ALGOL OPERATION SHEET

DESCRIPTION

Title ....

Programmer .... Tel. No. ....

Date submitted ....

Expected time .... Max. time ....

Operator ....

Date run ....

Actual time ....

TRANSLATION

1. Type of translation
   (*a*) Normal ☐ (*b*) owncode ☐ (*c*) omitted ☐
   (*d*) check for syntactic errors only ☐
2. Checks required (B-digit) ☐
3. Number of program tapes ☐
4. Input precompiled package with title
   .... ☐
5. Program contains 'precompile' ☐
6. Continue translation after 'precompile' ☐

Operator's comments

RUNNING

7. Program is
   (*a*) Normal ☐ (*b*) own code ☐ (*c*) dumped ☐
8. Use of equipment (*a*) reader 2 ☐
   (*b*) punch 2 ☐
9. Checks required (B-digit) ☐
10. Number of data tapes (*a*) reader 1 ☐
    (*b*) reader 2 ☐
11. Manual 'dump' ☐
12. Program contains 'wait' ☐
13. Program contains 'dump' ☐
14. Continue running after 'dump' ☐
15. Continue running after errors
    (*a*) Floating point overflow ☐
    (*b*) Others ☐
16. Manual restart ☐

SPECIAL INSTRUCTIONS

503 TECHNICAL MANUAL

VOLUME 2: PROGRAMMING INFORMATION
PART 1: PROGRAMMING SYSTEMS
SECTION 4: AUTOCODE

The contents of this section are liable to alteration without notice

March 1964
Reprinted with minor alterations
First printed January 1964

ELLIOTT
SCIENTIFIC COMPUTING DIVISION
Elliott Brothers (London) Limited
Elstree Way, Borehamwood, Hertfordshire, England

CONTENTS LIST

THE 503 AUTOCODE

1. INTRODUCTION

1.1. General

The 503 Autocode is an automatic coding system intended to be used as an aid in preparing programs for the Elliott 503 Computer. It has been designed primarily for the scientific user on the 803 and is provided as a 503 program primarily for the use of persons familiar with the 803 Autocode.

The 503 Autocode system is under the control of the Reserved Area Program (R.A.P.) (see 2.2.1), but Autocode tapes 1 and 2 (the translator and the functions and ancillaries tape respectively) are fixed, i.e. they always go into the same positions in the store. Tape 1 goes into the bottom of the store (i.e. the low addressed end of the store), tape 2 at the top of the free store (the store not used by the R.A.P.).

1.2. Method

The calculation to be carried out is first stated as a sequence of instructions written in a mnemonic code which closely resembles ordinary mathematical notation. This mnemonic program is then transcribed to punched tape.

The Autocode Translator (Tape 1) and the Functions and Ancillaries (Tape 2) are then placed in the 503, and the computer is used to translate the mnemonic program and run it, that is, to carry out that particular calculation. This method of translating a program and running it immediately is termed Load and Go. While the Autocode Translator and the Functions and Ancillaries are in the 503 it is permissible to translate and run as many programs as is required, one after another. Each program translated Load and Go overwrites the previous one.

Alternatively, it is possible to carry out the translating and running processes separately. In this case the Autocode Translator outputs the translated program on punched tape, and this translated program together with the Functions and Ancillaries is placed in the computer whenever it is desired to run the program. If this method is being used it is possible to specify a limited part of the computer store in which the program is to be run (translating onto tape for a limited part of the store).

Translated programs are output in a special relocatable binary, suitable for input under the R.A.P. Each one is preceded by a relocatable binary input routine, which the R.A.P. places in a fixed position at the top of the free store.

1.3. Compatibility with 803 Autocode

All programs written in 803 Autocode can be translated and run using the 503 Autocode if they are first re-punched onto 8-hole tape. The character @ does not appear in the 8-hole telecode and must be replaced by [. The value of n in OUTPUT n instructions should also be altered because of the change of code. The VERIFY facility is omitted in 503 Autocode because it uses up a lot of extra storage locations and is quite complicated to use. The autocode post mortem can be used to output the current values of specified variables (see Chapter 26). If the word VERIFY occurs in a 503 Autocode program then it is ignored by the translator.

The repunching of the mnemonic tape can either be done by hand from the program sheets or through the computer using the 'Autocode 5 to 8' (see 2.2.3.6.).

1.4. Running 803 Autocode programs on the 503

It is possible to run 803 Autocode programs on the 503 provided 5-hole paper tape equipment and 803 Autocode Tapes 1 and 2 are available. The program is run under control of the '803 Operator' (see 2.2.3.1.) and is subject to the limitations listed in that description.

1.5. Program Sheets

The program sheets for the 503 Autocode are not published.

2. OPERANDS

2.1. General

An operand is either a variable, or a constant, and it may be a floating-point quantity or an integer. There are therefore four classes of operands:

(i) Floating-point variables
(ii) Floating-point constants
(iii) Integer variables
(iv) Integer constants

It is not necessary to know exactly what the terms floating-point quantity and integer imply, provided it is remembered that:

(a) Floating-point quantities may range in magnitude from 4.3×10^{-78} to 5.8×10^{76} approximately, or be zero. Any quantity which, as a result of some arithmetic step, would have less magnitude than 4.3×10^{-78} appears as zero. If any attempt to produce a result having greater magnitude than 5.8×10^{76} is made, floating-point overflow will occur. This will cause an Error Interrupt and control will be transferred to the R.A.P. The method of representation is described in 1.2.1.
(b) Integers are numbers which may take on any integral value between $\pm$274 877 906 943.
(c) It is not permissible for floating-point quantities to enter into arithmetical operations involving integers, and vice versa.

2.2. Variables

Any letter of the alphabet, both upper and lower case, can be used to represent any variable. The programmer must decide which letters shall represent floating-point variables and which shall represent integer variables throughout any one program. It is possible to set lists of variables to their initial values (see Chapter 11).

2.3. Constants

The constants which occur in the calculation should be written in the program in normal decimal notation and will automatically be treated as floating-point constants or as integer constants according to the type of variable with which they are associated.

In instructions involving one or more integer variables, a decimal point may not appear. Constants of the form 5 and -720 may appear, 5 must be written as such; the forms 5. and 5.0 are not permitted.

In instructions involving one or more floating-point variables, constants of the form 5, 5.0, -720, -.003, and 241.47 may be used. Normal notation must be adhered to : 1.25×10^{-6} should be written .00000125 (the form 1.25/-6 or 1.25_{10}-6 must not appear on a program sheet).

There must not be more than 12 digits altogether. If there are 12 digits, their 'integer magnitude' must be less than 274 877 906 944. (By 'integer magnitude' is meant the magnitude of the number which would be seen if the decimal point were deleted.)

2.4. Conventional Allocation of Letters in this Text

To reduce the amount of explanation needed, the following conventions are adopted throughout the remainder of this text.

A, B, C and D	represent	Floating-point variables
I,J,K and L	represent	Integer variables
l, m and n	represent	Positive integer constants
p, q and r	represent	Any integer constants
x, y and z	represent	Floating-point constants

2.5. Suffices

Positive and zero integers may be used as suffices to any type of variable. A suffix may be written in any of several forms; these involve integer constants and/or variables. To take account of the fact that Flexowriter equipment has no multi-level printing facility, it is customary to write suffices on the same level as the letters which they qualify, that is to write A4, not A_4.

The table below shows the forms permitted. On the first four lines it will be noted that two or three different forms are grouped together; these are alternative ways of specifying the same quantity.

Floating-point Variables	*Integer Variables*	*Type of Suffix*	
A or A0 or A(0)	K or K0 or K(0)	Numerical	Simple
An or A(n)	Kn or K(n)		
AI or A(I)	KI or K(I)		
A(I+n) or A(n+I)	K(I+n) or K(n+I)		
A(I−n)	K(I−n)		
A(n−I)	K(n−I)	Complex	
A(I±J)	K(I±J)		
A(I±nJ)	K(I±nJ)		
A(mI±J)	K(mI±J)		
A(mI)	K(mI)		
A(mI±n)	K(mI±n)		
A(m±nI)	K(m±nI)		

In the examples below, whenever A, B, C, D, I, J, K or L appear as variables, any of the above forms may be substituted unless otherwise stated. Suffices may not themselves have suffices. In particular, A(mI) and A(Im) are both interpreted as $A_{m \cdot I}$.

Note carefully that suffices should never be negative. Errors will occur if negative suffices are used.

2.6. Brackets

The combination of (and) is used to surround suffices; and for no other purpose. They are obligatory where the suffix contains more than one component. Thus A(I+4) corresponds to the mathematical notation a_{i+4}; there is no parallel in the notation of this autocode to the mathematical form (a+b)(c+d)

2.7. Functions

Existing function names contain only upper case letters (A to Z) but if additional functions are added to the translator they may contain both upper and lower case letters.

3. STYLE OF WRITING

Instructions are written in columns. Spaces are left after words and quasi-words, but not elsewhere. These points are dealt with more fully in Chapter 20, but to give a preliminary idea of what a simple program in autocode looks like, the instructions needed for a complete program to read in values of A and B, form and print $\sqrt{A^2+B^2}$ are:

```
SETV ABC
SETF SQRT
SETR 1
1)READ A
READ B
A=A*A
B=B*B
C=A+B
C=SQRT C
LINE
PRINT C
STOP
START 1
```

4. ARITHMETIC INSTRUCTIONS

Arithmetic instructions are of the form

A = an expression involving floating-point quantities

or

I = an expression involving integers

e.g.

A=B+C

I=J—2

and mean 'evaluate the expression on the right and then set the variable on the left to this value'.

The range provided is

(a) Setting to a value:	A=B	I=J
(b) Addition:	A=B+C	I=J+K
(c) Subtraction:	A=B—C	I=J—K
(d) Multiplication:	A=B*C	I=J*K
(e) Division:	A=B/C	—

In which

(a) B and C may be replaced by constants, or by A, or be identical.
For example: A=2*A and A=A+A are alternative methods of doubling A.

(b) J and K may be replaced by integer constants, or by I, or be identical.
For example: I=I+1 increases the value of I by 1.

(c) A—sign may follow immediately after the = sign.
For example: A=—A negates A.

(d) The symbol * is used for multiplication to avoid confusion with the letter X.

(e) Division is indicated by /.

(f) Note that division of integers is not permitted.

Examples of permitted instructions:

```
J=3
I=—4*J
C(I+2J) =—AI—B(2J)
A(I+6) =—B(I+6)/4.275
```

5. FUNCTION INSTRUCTIONS

Most function instructions are of the form:

A = FUNCTION B   or A = FUNCTION x

and again mean 'evaluate the expression on the right and then set the variable on the left to this value'. A − sign may follow the = sign. B may be identical with A, but must not be preceded by a − sign.

The following functions are used as above, that is, with a floating-point variable on the left and a floating-point quantity as argument:

SIN	sine	In which the unit is the straight angle of π radians or 180°, and the angle must have less magnitude than 268 435 456. e.g. SIN 0.25 = sin 45° = $1/\sqrt{2}$
COS	cosine	
TAN	tangent	
ARCTAN	arctangent	In which the result is in straight angles, and lies between $\pm\frac{1}{2}$
LOG	natural logarithm, that is, to base e	
EXP	exponential	
SQRT	square-root	(SQRT 0=0)
INT	integral part	
FRAC	fractional part	
MOD	modulus	

The integral part of a negative number is minus the integral part of its modulus, and its fractional part is such that:

integral part + fractional part = whole

For example:

INT −3.4 is −3
FRAC −3.4 is −.4

Certain function instructions are provided which involve integers. These are of the form:

I=INT A	Take integral part of A, convert it to integer form, and set I to this value.
A=STAND I	Take I, convert it to floating-point form, and set A to this value. (The technical term for this process is standardisation.)
I=MOD J	Make I equal to modulus of J.

Additional functions may be added to the above range: see Appendix 4.

6. REFERENCE NUMBERS

In general, the computer performs the steps in a calculation in the sequence in which the instructions are written down, and for this reason it is not necessary to number all the instructions in a program. However, occasions arise in which it is desirable that the computer should break sequence and 'jump' to some other part of the program. Also, it must be possible to specify which instruction is to be obeyed first, and to program the computer to stop when it has finished its calculations.

The instructions provided to achieve the above are detailed in Chapters 7, 8, 9, 10 and 12. Their form is such that it must be possible to make reference to one instruction in another. All instructions to which reference is so made must be given different reference numbers. In writing the program, reference numbers appear on the left, each followed by a) thus:

```
4)A=B+C
12)I=INT B
```

As a rule, if there are n reference numbers in a program, they will be the integers 1, 2 ...... n, but they need not occur in that sequence. If, as a result of some afterthought, it becomes necessary to delete one or two reference numbers it is not essential to close the gaps.

7. SETTING INSTRUCTIONS AND START

7.1. General

To enable the Autocode Translator to construct the translated program correctly, it must receive certain general information before it starts translating the mnemonic instructions. Up to four setting instructions, which appear at the beginning of a program, are used to do this.

In addition, a START instruction appears at the end. This has two roles. Firstly, it specifies which of the instructions of the program is to be obeyed first. (Afterthoughts are often placed at the end, and it is quite common for the instruction which is first to be obeyed to be nearly the last one written and to have a high reference number.) Secondly, it is written after all other mnemonic instructions, to indicate to the translator that there are no more instructions to be translated.

7.2. Details of the Setting and START instructions

(a) SETS

The SETS instruction specifies the letters to be used as integer variables. It may be omitted if no integer variables occur in the program. The sequence in which the letters are written is immaterial.

After each letter one must insert, in brackets, a number indicating the highest value of any suffix attached to that letter in the program. If, for example, JI occurs in the program, one must take into account the highest value which I possesses at any time at which an instruction referring to JI is obeyed. If no suffices or only suffix 0 are used, no number is needed.

Thus:

```
SETS IJ(4)K
```

indicates that I or I0, J or J0, J1, J2, J3, J4 and K or K0 are or may be used in the program.

(b) SETV

SETV specifies the letters used to represent floating-point variables in exactly the same way as SETS specifies letters used for integer variables. It may be omitted if no floating-point variables occur in the program.

(c) SETF

SETF specifies the functions which are required in the calculation, and certain types of peripheral equipment. The abbreviation TRIG must be used to cover any combination of SIN, COS and TAN. The two functions MOD and STAND are always available, and need not be mentioned. The sequence in which the functions are written is immaterial. SETF may be omitted if no functions (other than STAND and MOD) are required, and if there are no instructions referring to more than one reader and/or punch (see Chapter 17).

Thus:

```
SETF TRIG SQRT LOG EXP
```

is suitable for a calculation requiring some or all of: standardised forms, moduli, sines, cosines, tangents, square roots, logarithms and exponentials.

(d) SETR n

This instruction must always be used: n specifies the highest reference number used in the program.

(e) START m

This instruction must always be used: m specifies the reference number of the first instruction to be obeyed.

7.3. Positioning of Setting and START Instructions

The setting instructions are written in a group at the beginning of the mnemonic program, with SETR last. The mnemonic instructions should follow immediately below the SETR instruction, the one on the line below the SETR necessarily having a reference number.

The START instruction is written at the end of the mnemonic program, that is after the last instruction to be translated, which will generally be JUMP, EXIT or STOP.

8. JUMP INSTRUCTIONS

8.1. Unconditional Jumps

JUMP [n (or JUMP [I)

Means 'break sequence and obey next the instruction whose reference number is n (or .... is equal to the current value of I), and then proceed sequentially from there'.

In the form JUMP [I, the I may have a numerical suffix.

8.2. Conditional Jumps

JUMP IF A=B [n (or ...... [I)
JUMP UNLESS A=B [n (or ...... [I)

The first of these means 'If A=B, act as for JUMP [n (or I), but if A≠B, do nothing'. The second has the reverse effect, that is 'If A=B, do nothing, but if A≠B, act as for JUMP [n (or I)'.

In the form JUMP IF (or UNLESS) A=B [I, I may have a numerical suffix.

The condition A=B in the above examples may be replaced by any permitted arithmetic instruction form as given in Chapter 4 or any permitted function instruction form as given in Chapter 5. Furthermore, the = sign may be replaced by > or <.

The following are a few examples of the vast number of different conditions that can be stipulated in JUMP IF and JUMP UNLESS instructions:

A=B	A<0	A>4.35
A=B/2.5	A<B+C	A>4*B
A=MOD B	A<SIN B	A>MOD B
I=J	I<0	I>7
I=J*3	I<MOD J	A>STAND I
	A(I+2J)<-BI-C(2J)	

9. SUBROUTINE ENTRY AND EXIT

SUBR n

means 'jump to the subroutine which starts with the instruction having reference number n'.

SUBR I

means 'jump to the subroutine which starts with the instruction having reference number equal to present value of I'. I may have a numerical suffix.

EXIT

means 'return to the main program or superior subroutine at the instruction immediately after the SUBR which last caused entry to this subroutine'. Where the entry to a subroutine is by SUBR instruction, the exit must always be by EXIT, not by a jump or conditional jump.

There may be subroutines within subroutines up to six in depth, i.e. at no point may more than 6 SUBR instructions have been obeyed without a corresponding EXIT instruction.

10. STOP, WAIT AND SELECTED STOP

10.1. STOP

The STOP instruction is translated as a transfer to the R.A.P. which, when obeyed at run time, causes the word END to be output on the output writer. The R.A.P. then waits for a message.

There may be more than one STOP instruction in a program which has several branches, corresponding to the end of calculation or the reaching of dead-end positions.

10.2. WAIT

This instruction is written in a program at any point at which it is desired that the calculation should be halted until the operator has had a chance to take some special action or other. (It has the effect of stopping the program until the setting on the N2 buttons of the word generator has been changed from odd to even or vice versa.)

10.3. SELECTED STOP

As an additional aid in finding program faults this facility has been incorporated to cause the calculation to stop at some specified reference number n. That number should be set on the N1 buttons before the program is entered.

When the instruction with reference number n is reached, control is transferred to the R.A.P. to output the word 'STOP' on the typewriter and to await a message. Before a message is typed the setting on the N1 buttons should be either

(i) cleared if no more selected stops are required

or (ii) changed to a new value m (say). In this case when calculation is resumed it continues until the instruction with reference number m is reached, etc.

or (iii) left equal to n. In this case when calculation is resumed, the instruction with reference number n is obeyed and the program continues until that instruction is reached again.

Possible messages are

(a) CONT. Control is transferred back to the program by the R.A.P. and the calculation is resumed at the instruction with reference number n.

(b) 16. This has the effect of restarting the program.

(c) 7. (load and go) or 5. or 6. (on to tape). If the load and go facility is being used then the program could be retranslated, or a new program could be translated. It is necessary to re-input the translator before translating on to tape.

(d) If the post mortem routine is in store, then it can be entered by typing its name AUTOPM. The current values of variables specified on a control tape are printed on a specified output device. The post mortem is fully described in Chapter 26.

(e) If the post mortem routine is not in store, it may be input by placing it in the tape reader and typing IN. The post mortem is entered as described in Chapter 26.

11. INITIAL VALUES

SET A(n:m) = x, y, z, ... a list of (m−n+1) floating-point constants

SET I(n:m) = p, q, r, ... a list of (m−n+1) fixed-point constants

By means of these mnemonic instructions it is possible to set lists of variables to their initial values. The values in the list must be separated by , or L̲ or S̲ or T̲. If two separators occur the second one is ignored.

The number of constants in the list is only limited by the setting instructions SETV A(i) or SETS I(i), where i $\geqslant$m. There is no check on this limit in the translator.

Mnemonic instructions which set variables to their initial values must not be confused with Setting Instructions (see Chapter 7). They may occur at any point in a program after the SETR instruction, but when the program is translated they come into effect immediately after the Setting Instructions, i.e. before the actual calculation commences.

Example

```
SETS I(6)
SETR 1
1)  .
    .
    .
SET I(1:4) = 5, 6, 7, 8
    .
    .
```

The effect of this instruction is to set the initial values 5, 6, 7 and 8 in I(1), I(2), I(3) and I(4) respectively before the first instruction (in this case that with reference number 1) is obeyed.

It must be understood that a program including such orders can only be re-run correctly, without re-input, if the initial values remain unaltered.

The effect of writing a reference number before a SET instruction is that on translation the reference number will appear before the next instruction which is not itself a SET instruction.

Example

The following pairs of instructions give identical translations:

```
3)SET I(0:2)=0,1,2          SET I(0:2)=0,1,2
  A=6                       3)A=6
```

12. VARY AND CYCLE INSTRUCTIONS

The VARY and CYCLE facilities are provided in order that the computer may be caused to carry out a set or loop of instructions several times. A loop is commenced with one of six different types of instructions involving the word VARY or the word CYCLE: this specifies that the loop shall be repeated several times with a particular variable, the counting variable, at different values. A REPEAT instruction ends the loop. To each VARY or CYCLE instruction there must be one, and only one, REPEAT instruction.

No reference numbers have to be given to VARY, CYCLE or REPEAT instructions unless desired. However, greater advantage can be taken of the trace facility described in Chapter 18.2 if each REPEAT instruction is numbered.

Throughout Chapters 12.1 to 12.4 below, the word (LOOP) represents 'any set of instructions which does not alter any of the variables mentioned in the VARY or CYCLE instruction'. It is also assumed that the variables to the right of the = sign in the VARY or CYCLE instruction are not identical with that on the left. More complex cases, to which these conditions do not apply, are discussed in Chapters 12.6 and 12.7.

12.1. VARY Instructions

(a)
```
VARY I=J:K:L
(LOOP)

REPEAT I
```

The loop is performed with $I = J, J+K, J+2K, \ldots, J+(L-1)K$, that is L times in all.

I may be of any permitted integer variable form.

J and K may be preceded by − signs, or may be replaced by integer constants. L must be positive; it may be replaced by a positive integer constant. J, K and L may have simple suffices only.

(b) VARY A=B:C:L
(LOOP)

REPEAT A

The loop is performed with A = B, B+C, B+2C,...... B+(L−1)C, that is L times in all.
A may be of any permitted floating-point variable form.
B and C may be preceded by − signs, or may be replaced by constants.
L must be positive; it may be replaced by a positive integer constant.
B, C and L may only have simple suffices.

12.2. CYCLE Instructions, Type 1

(a) CYCLE I=J:K:L
(LOOP)

REPEAT I

The loop is performed with I = J, J+K, J+2K, ..., L that is $\left(\frac{L-J}{K} + 1\right)$ times in all. It is essential that $\frac{L-J}{K}$ is integral, and $\geqslant 0$.

I may be of any permitted integer variable form.
J, K and L may be preceded by − signs, or may be replaced by integer constants. They may have simple suffices only.

(b) CYCLE A=B:C:D
(LOOP)

REPEAT A

The loop is performed with A = B, B+C, B+2C, ..., D.

$\frac{D-B}{C}$ must be $\geqslant 0$, but provided it is $> \frac{1}{2}$ it need not be exactly integral. The value of A in the final performance is adjusted to be exactly equal to D, and the number of times the loop is performed is the nearest integer to $\left(\frac{D-B}{C} + 1\right)$. Thus:

(i) For CYCLE A=1:1:3.1
the loop is performed with A = 1, 2, 3.1

(ii) For CYCLE A=1:1:3.9
the loop is performed with A = 1, 2, 3, 3.9

(iii) For CYCLE A=1:1:3.5
the loop is performed with A = 1, 2, 3.5
A may be of any permitted floating-point variable form. B, C and D may be preceded by − signs, or may be replaced by constants. They may only have simple suffices.

12.3. Comparison of the VARY and CYCLE Type 1 Facilities

(a) With an integer counting variable
CYCLE I=J:K:L
and
VARY I=J:K:n
are identical if (n−1)K = L−J, so the choice is a matter of convenience.

(b) With a floating-point counting variable
CYCLE A=B:C:D
and
VARY A=B:C:n
where (n–1) C = D–B, are theoretically identical. However the practical points arising out of the use of floating-point arithmetic influence the choice. These are:
(i) The VARY version is much faster, but, unless the values of B, C and D are all exactly expressible in the binary notation used in the computer, the final value of A, which is formed by successive addition of C to B, may not be identically equal with the desired final value D.
(ii) The CYCLE version guarantees that the final value of A is identical with the desired final value D, but is performed more slowly.

12.4. CYCLE Instructions, Type 2

(a) CYCLE I=p,q,r,. . . .
(LOOP)
REPEAT I
The loop is performed with I = p, q, r, ...
There are no restrictions on the form of I.
p, q, r etc. are integer constants, and there may be any number of them. There must be no comma after the last number in the sequence.

(b) CYCLE A = x, y, z, ...
(LOOP)
REPEAT A
The loop is performed with A = x, y, z, ...
There are no restrictions on the form of A.
There must be no comma after the last number in the sequence.
x, y, z etc., are constants, and there may be any number of them.

12.5. Loops within Loops

When translating load and go CYCLE Type 1, Type 2 and VARY may not occur inside each other in any combination to more than eight in depth.
When translating and running separately there are no restrictions on the depth of nested loops.

12.6. Altering the Counting Variable within the Loop

If one or more instructions to alter the counting variable are included in the loop, the effects stated in Chapters 12.1, 12.2 and 12.4 will be modified as below.

(a) With VARY
The loop is performed L times, the value of the counting variable being increased by the increment (K or C) between the end of each performance and the beginning of the next, but not after the last.

(b) With CYCLE Type 1
(i) With an integer counting variable.
If the value of the counting variable when the REPEAT instruction is reached is exactly equal to the final value , L, cycling will cease. Otherwise, the counting variable will be increased by the increment and the loop repeated. See note.

(ii) With a floating-point counting variable.
The conditions under which the value of a floating-point counting variable can, with safety, be altered within a CYCLE Type 1 loop, are so restrictive that it is more convenient to arrange the program in such a way that the need is avoided than to comply with them. See note

Note: If, due to the intervening instructions, the value of the counting variable when the REPEAT instruction is reached is never equal to the final value, cycling will continue indefinitely, unless some other provision, such as the incorporation of a JUMP IF instruction in the loop, is made.

(c) With CYCLE Type 2
The loop is performed as many times as there are values in the CYCLE instruction, starting each performance with the counting variable at each of the values in turn.

12.7. Other Special Effects

By writing VARY and CYCLE Type 1 instructions and loops in which the variables to the right of the = sign are either (i) identical with the counting variable or (ii) altered within the loop, one may obtain various special effects. A few examples are given here:

(a)
```
VARY I=I:I:n
(LOOP)
REPEAT I
```
In the first performance of the loop, I retains whatever value it possesses when the VARY instruction is reached. In subsequent performances it is increased by itself. We may therefore say, writing I_0 for the initial value of I, that the loop is performed with $I = I_0, 2I_0, 4I_0, \ldots . 2^{n-1}I_0$.

(b)
```
B=0
CYCLE A=1:B:16
(LOOP)
B=B+1
REPEAT A
```
The loop is performed with A = 1, 2, 4, 7, 11, 16.

(c)
```
CYCLE I=J:K:L
(LOOP containing an instruction altering L)
REPEAT I
```
Effect as in Chapter 12.6(b)(i)

(d)
```
VARY A=B:C:L
(LOOP containing an instruction altering L)
REPEAT A
```

Cycling continues until the number of times the loop has been performed is exactly equal to the value of L at the time the REPEAT instruction is reached.

13. OUTPUT THE ALPHABET TABLE AND MODIFY

13.1. Facility to output the alphabet table

On completion of translation of an autocode program.
(a) type 1358. to output a–z and A–Z

(b) type 1370. to output A–Z only.

The output is on the output writer and will be of the following form:

```
A = 6993
B = 6992
C =
D = 6989
etc.
```

If there is no number on the right-hand side of an equals sign then that letter has not been set by SETS OR SETV.

13.2. MODIFY or modification of mnemonic tapes

This facility is designed to reduce the amount of tape copying that is required when either program or tape errors are discovered.

Orders between reference numbers can be re-written and translated at the end of the mnemonic program, but before the START n instruction has been translated.

Thus, in order to use this facility, the original tape should end

```
SL
blanks
START n
```

Note: Reference numbers included in SETR but previously unallocated may be freely attached to the modifying instructions.

The modification tape is punched as follows:

```
MODIFY
m)  ----------    the modified instructions.
    ----------    N.B. There must be no other reference number between m and p
    ----------    except previously unallocated ref. nos. Comments are not allowed
    ----------    after MODIFY.
p)  SL
    blanks
```

After the p) it is permitted to write MODIFY L instead of S L if more than one modification is required.

After the p) of the last modification, it is permitted to write START n L instead of S L.

If a VARY or CYCLE instruction appears in a modification then the corresponding REPEAT *must* appear in the same modification and vice versa. Two examples of this are given below.

Example: *Original program* *Modification*

```
      .
      .
      .
 24) VARY J=1:1:N                   blanks
     READ CJ                        MODIFY
     READ J                     24) VARY J=1:1:N
 20) Z=0                            READ CJ
      .
      .
      .                             X=X+CJ
      .
  2) Y=0                            REPEAT J
     VARY J=1:1:N               20) MODIFY
     JUMP UNLESS Y>ZJ [3         2) Y = 0
     Y=ZJ                           VARY J=1:1:N
```

```
3) REPEAT J                JUMP UNLESS Y<ZJ [3
     .                       Y=ZJ
     .                    3) MODIFY
     .                    3) REPEAT J
     .                          .
4)   .                          .
                                .
                                .
                          4) S L
                             blanks
```

When the program is translated, the translation of both sets of orders between 24) and 20), and 2) and 3) will be present, but the reference table will be altered so that at run time only the modified orders will be obeyed.

It is advisable, once the program is working correctly, to make a correct version of the mnemonic tape.

Method of Use

(a) Translate the tape in the normal way until it stops in a keyboard loop just prior to the START n instruction. (If the tape stops on an error which is known to be corrected by a modification tape then change the sign digit to carry on translating, ignoring the error stop.)

(b) Remove the main tape from the tape reader and replace it with a modification tape. Change the sign of the word generator to cause translation to recommence.

(c) When all the modification tapes have been read in a similar manner, replace the main mnemonic tape in the reader at the blanks before START n and change the sign of the word generator for the translation to be completed.

(d) Run the translated program in the usual way.

Modification of an autocode program after completion of translation

Load and Go only

When translating and running an autocode program load and go, it is possible to translate a modification to the program either after the START instruction has been translated or after running the program (provided that the translator has not been overwritten), by using the MODIFY facility.

The modification tape (or tapes) should be prepared as already described and it must be translated by placing it in the tape reader and typing 17. . If there is more than one modification tape, change the sign of the word generator to continue translating the modifications.

The START instruction must be translated after the last modification. It may be either the START instruction on the original mnemonic tape or a new START instruction which specifies a different starting reference number from the original one.

14. INPUT INSTRUCTIONS

14.1. READ A and READ I

both mean 'read the next number on the input tape, and set the specified variable to the value read'.

In addition to its main purpose, that of reading numbers into the computer, the READ function also provides facilities by means of which the course of the program can be controlled by triggers and stops punched on the data tape. Additionally, if it is desired that any explanatory notes should appear on the output, these can be punched on the data tape in the form of labels, and will be copied on to the output tape. For details, see Chapter 21.

14.2. INPUT I

means 'read the next character on the input tape, and set the specified integer variable to its numerical value'. I may only have a simple suffix.

The INPUT function is useful for data tape control of the program, being primarily intended for reading control characters.

The 8-channel Paper Tape Code is as follows:

00000.000	0	$\underline{\text{R}}$ (blank)	01010.000	32	;	10010.000	64	$\underline{\text{S}}$	11000.000	96	?
00010.001	1		01000.001	33	A	10000.001	65		11010.001	97	a
00010.010	2	$\underline{\text{L}}$	01000.010	34	B	10000.010	66		11010.010	98	b
00000.011	3	$\underline{\text{P}}$	01010.011	35	C	10010.011	67		11000.011	99	c
00010.100	4	$\underline{\text{T}}$	01000.100	36	D	10000.100	68		11010.100	100	d
00000.101	5	$\underline{\text{B}}$	01010.101	37	E	10010.101	69		11000.101	101	e
00000.110	6		01010.110	38	F	10010.110	70		11000.110	102	f
00010.111	7		01000.111	39	G	10000.111	71		11010.111	103	g
00011.000	8	(	01001.000	40	H	10001.000	72		11011.000	104	h
00001.001	9	)	01011.001	41	I	10011.001	73		11001.001	105	i
00001.010	10	,	01011.010	42	J	10011.010	74		11001.010	106	j
00011.011	11	£	01001.011	43	K	10001.011	75		11011.011	107	k
00001.100	12	:	01011.100	44	L	10011.100	76	$\underline{\text{H}}$	11001.100	108	l
00011.101	13	&	01001.101	45	M	10001.101	77		11011.101	109	m
00011.110	14	*	01001.110	46	N	10001.110	78		11011.110	110	n
00001.111	15	/	01011.111	47	O	10011.111	79		11001.111	111	o
00110.000	16	0	01100.000	48	P	10100.000	80		11110.000	112	p
00100.001	17	1	01110.001	49	Q	10110.001	81		11100.001	113	q
00100.010	18	2	01110.010	50	R	10110.010	82		11100.010	114	r
00110.011	19	3	01100.011	51	S	10100.011	83		11110.011	115	s
00100.100	20	4	01110.100	52	T	10110.100	84		11100.100	116	t
00110.101	21	5	01100.101	53	U	10100.101	85		11110.101	117	u
00110.110	22	6	01100.110	54	V	10100.110	86		11110.110	118	v
00100.111	23	7	01110.111	55	W	10110.111	87		11100.111	119	w
00101.000	24	8	01111.000	56	X	10111.000	88	[	11101.000	120	x
00111.001	25	9	01101.001	57	Y	10101.001	89	]	11111.001	121	y
00111.010	26	10	01101.010	58	Z	10101.010	90	$_{10}$	11111.010	122	z
00101.011	27	11	01111.011	59		10111.011	91	<	11101.011	123	
00111.100	28	=	01101.100	60		10101.100	92	>	11111.100	124	
00101.101	29	+	01111.101	61		10111.101	93	↑	11101.101	125	
00101.110	30	—	01111.110	62	$\underline{\text{V}}$	10111.110	94	~	11101.110	126	$\underline{\text{U}}$
00111.111	31	.	01101.111	63		10101.111	95	%	11111.111	127	$\underline{\text{E}}$

Abbreviations

$\underline{\text{B}}$	Backspace	$\underline{\text{T}}$	Tabulate
$\underline{\text{E}}$	Erase	$\underline{\text{H}}$	Stop Code (Halt)
$\underline{\text{L}}$	New Line	$\underline{\text{P}}$	Paper Throw
$\underline{\text{R}}$	Run Out	$\underline{\text{U}}$	Underline
$\underline{\text{S}}$	Space	$\underline{\text{V}}$	Vertical Bar

15. PRINT INSTRUCTIONS

15.1. General

A PRINT instruction has the meaning 'punch on the output tape, or print on the output writer,† the characters corresponding to the value of the specified variable in the specified mode'. However, it is more convenient to speak in terms of the characters which will be printed when the output tape is interpreted: this method of description will therefore be used.

† Henceforth the verb 'display' will be used for 'print out on the control typewriter'.

When a PRINT instruction is obeyed, a negative number is preceded by –, a positive number by $\underline{S}$. Non-significant leading 0's are replaced by $\underline{S}$'s. All numbers are followed by $\underline{S}$.

No page layout characters are punched except when the variable to be printed is too great for the mode of printing specified: the details of this are given below.

Although a floating-point quantity may be printed to more than 9 places, only the first 9 significant decimal digits are accurate. If the form in (c) below is used, there can be no purpose in making $m > 9$.

15.2. Detail

(a) PRINT A,m:n or PRINT A,I:J
A is printed with m or I digits before and n or J digits after the point. If n or J is zero, then the printing of the decimal point is suppressed. If m or I exceeds 9 then m is set = 9 and n = 3.

(b) PRINT A,m or PRINT A,I
A is printed with m or I digits altogether with the decimal point in the appropriate position. There are $(m + 3)$ or $(I + 3)$ characters altogether.
If $| A | < .5 \times 10^{-m}$ or if $| A | < .5 \times 10^{-I}$
the result appears as zero.

(c) PRINT A,m/ or PRINT A,I/
A is printed as a decimal fraction of m or I digits, followed by a $_{10}$ and a 2 digit exponent expressed as $\underline{S}$ or – and a two-digit number. Altogether $(m+8)$ or $(I+8)$ characters are printed. If m or I exceeds 9, then it is set to 9.

(d) PRINT A
A is printed in the same mode as the last floating-point variable printed. If no floating-point variables have yet been printed, the mode used corresponds to PRINT A,8/.

(e) PRINT I,m or PRINT I,J
I is printed as an m– or J– digit integer. A total of $(m+3)$ or $(J+3)$ characters is produced. m or J must not be greater than 12.

(f) PRINT I
I is printed in the same mode as the last integer printed. If no integers have yet been printed, the instruction is obeyed as for PRINT I,4.
In all cases I and J may have simple suffixes when used as print parameters.

15.3. Errors

(i) If A is in non-standard form then 'PRINT ERROR' is displayed. When the sign of the word generator is changed, ? is output on the current output device and zero is printed in the specified mode.

(ii) If I is too great for the mode of printing specified then the print routine standardises I and prints it as a floating point number in form (c) (A,m/) unless this upsets the page layout. In this case,) is output. If A is too great for the mode of printing specified then it is printed in the form (c) (A,m/) unless this upsets the page layout. In this case,) is output. If n is the total number of characters called for, then $m=n-6$ if $n > 6$. If $n \leqslant 6$ then the closed bracket) is preceded by $n-1$ spaces.

16. MISCELLANEOUS OUTPUT INSTRUCTIONS

16.1. TITLE

This instruction is used to cause a heading or other explanatory note to be printed.

All characters which appear on the mnemonic program tape between the $\underline{S}$ or $\underline{L}$ which follows the word TITLE and the next $\underline{R}$ or ; are punched on the output tape each time the TITLE instruction is obeyed. Care should be taken, in writing a TITLE instruction, to ensure that all necessary page layout characters and the final ; are specifically indicated. See Appendix 6.

N.B. A title may not include a ;.

16.2. LINE

Is used when it is desired to print the next number on a new line. It causes $\underline{L}$ to be punched.

16.3. LINES n or LINES I

Similar to LINE, but n or I $\underline{L}$'s are punched.

16.4. SPACES n or SPACES I

Cause n or I $\underline{S}$'s to be punched.

16.5. OUTPUT n or OUTPUT I

Cause the character with numerical value corresponding to the integer n or I to be punched. See Table in Chapter 14.2.

17. READER/PUNCH FUNCTION

17.1. General

The 503 is equipped with two tape readers, two tape punches, an input typewriter and an output writer. The instructions given below make it possible for a program written in 503 Autocode to read in data from either or both readers or from the typewriter, and to output results by means of either or both punches and/or on the output writer.

17.2. Setting Instructions

If it is intended to input other than from reader 1 and/or to output results other than on punch 1 then it is necessary to set READER and/or PUNCH as a function in the setting instruction SETF (see Chapter 7.2.(c)).

For more than one reader or for the input typewriter set READER.

For more than one punch or for the output writer set PUNCH.

For more than one reader and for more than one punch and/or the typewriter set either READER PUNCH or PUNCH READER.

17.3. Mnemonic Instructions

When an instruction READER n (or I) or PUNCH n (or I) is obeyed, the function checks that n (or I) lies in the range $0 \leqslant n$ (or I) $\leqslant 7$. If n (or I) lies outside the range, RPUNCH ERROR is displayed and the machine cycles in a keyboard loop. When the sign of the word generator is changed, the function sets n (or I) equal to 1 and continues.

Value of n	*Input/output device set*
1	tape reader 1, punch 1
2	tape reader 2, punch 2
3	input typewriter, output writer
4, 5, 6, 7	tape reader 1, punch 1

If n = 0, the function displays READ 0 or PUNCH 0 and reads a message from the typewriter of the form m. where $0 \leqslant m \leqslant 7$, i.e. at that point in the program the operator can choose the input/output device for all the following input/output orders until a further READER/PUNCH instruction is obeyed. If m = 1 then tape reader/punch 1 is set, etc. If, after PUNCH 0, m = 0 then printing is suppressed. If, after READ 0, m = 0 then RPUNCH ERROR is displayed and input continues from tape reader 1 when the sign of the word generator is changed.

The instruction TELEPRINTER has the same effect as PUNCH 3.

17.4. Notes

(a) The Translator sets Punch 1 and Reader 1 as the currently available mechanisms at the beginning of every Load and Go program and any translated program which has PUNCH and/or READER set in the SETF instruction.

(b) The mnemonic program tape(s) will always be read in from Reader 1.

(c) The translated program will always be output on Punch 1.

(d) All error indications occurring during the running of a program will be output on the output writer.

18. CHECK INSTRUCTIONS AND TRACE FACILITY

18.1. CHECK A or CHECK I

These instructions provide optional print-out of intermediate results. It is advisable to include them at suitable points in all long programs, to assist in detecting program errors. If the B-digit key on the word generator is kept depressed during the running of a program, they cause A or I to be printed on a new line, preceded by * and in the mode corresponding to PRINT A,8/ or PRINT I, 12. The checks are output on the current output device at run-time. If the B-digit key is not kept depressed, they are ignored.

Note, however that CHECK instructions are only included in the translated program if the B-digit key is kept depressed during translation. Thus, when a program has been tested and found correct, a new translated tape without CHECK'S can be prepared from the original mnemonic tape.

18.2. Trace Facility

If a program is not running its proper course, the trace facility can be used to determine at what point the error arises.

If 40 is set up and retained on the F2 keys of the word generator during the running of a program, the reference number of each numbered instruction obeyed is printed on a new line, followed by). The traces are output on the current output device at run-time.

Note, however, that the trace facility is only included in the translated program if 40 is set up and retained on the F2 keys during translation.

If a habit is made of numbering every REPEAT instruction, the trace facility can be used to determine how many times each CYCLE or VARY loop is obeyed.

19. BLOCKS OF MACHINE CODE PROGRAM

19.1. General

For the convenience of programmers who are familiar with the 503 machine code, it is possible to include blocks written in a style similar to the Symbolic Assembly code (see 2.2.2.) amongst the mnemonic instructions of a program written in autocode.

The first word of such a block must be preceded by [which itself may be preceded by a reference number and) if desired. The last word of the block must be followed by) on a new line on its own. It is not possible to give a reference number (or label) to a word in the middle of a block.

Absolute locations 12, 13 and 14 may be used as temporary working space within the block but not elsewhere in the program since their contents may be destroyed by one of the Autocode's own subroutines.

Entry to a machine code block may be made either by a mnemonic JUMP type instruction to the reference number of its first word, or in normal sequence whereby the machine code instruction immediately following [is obeyed after the mnemonic instruction preceding it. Similarly exit from the machine code block may be made by using function 40, 41, 42 or 43 to any instruction with a reference number, or in normal sequence from the last instruction before) to the first mnemonic instruction after it.

A machine code block may be entered by a SUBR instruction by giving its first word a reference number, and ensuring that its exit leads to an EXIT instruction.

19.2. Instructions

Instructions are written in a fashion similar to that used in the S.A. code and may contain addresses of four types:

(a) Absolute

```
30    13    :    55    5
```

(b) Relative to the first word of the block:

```
26    13    :    30    14,
```

(c) Referring to variables having numerical suffices, suffix zero being omitted if desired:

```
24    I     :    30    A4
```

(d) Reference numbers in the mnemonic part of the program, or at the start of other machine code blocks. These may only be used with functions 40 to 43:

```
43    3)    :    40    5)
```

Reference to variables having suffices which are not numerical may be made by using B-lined instructions. Provided such suffices are not complex, this is not at all involved. For example, to bring A(I + 2) in to the accumulator, write

```
00    I     /    30    A2
```

19.3. Constants

Constants may be fixed-point integers, unscaled fixed-point fractions of magnitude less than 1, or floating-point numbers. The usual capacity restrictions apply.

Fixed-point integers must consist of + or — followed by one or more decimal digits, without a decimal point.

Fixed-point fractions must consist of + or – followed by one or more decimal digits.

Floating-point numbers must consist of + or — followed by one or more decimal digits and a decimal point. Further digits may follow the point if needed. The forms +.125/12 and $+.125_{10}12$ are not permitted.

Examples	Fixed-point	−123	+.345
	Floating-point	−123.	+0.345

19.4. Floating-point Arithmetic

The functions specified in machine code blocks are obeyed directly, not interpreted. Thus 30 B : 04 A would be carried out in fixed-point fashion so that if A and B are non-zero floating-point quantities, the result would be nonsense.

19.5. Miscellaneous Points

Instructions must be written two to a line and zero instructions must be punched as 000, zero addresses as 0. The zero word should be punched as +0 L or 000 : 000 $\underline{L}$ and $\underline{S}$ and $\underline{T}$ should not appear in machine code blocks.

? cancels all characters since the last $\underline{L}$ or [. [and any reference number which precedes it can only be cancelled by over-punching.

20. PREPARING THE MNEMONIC PROGRAM

20.1. The Written Form-Main Features

The mnemonic program is written as a column or a number of columns of instructions, with the setting instructions at the top of the first and the START instruction at the bottom of the last.

An example of the type of program sheet used at Elliott Brothers (London) Limited, and a full description of its use, are given in Appendix 6. The following is an abridged description.

The REF. column is used for reference numbers. Instructions and remarks are written in the appropriate columns. The use of the FLOW column is at the discretion of the programmer. The END column is only used in special cases. Titles are written out in full detail and encircled.

In writing the instructions, a space should be left between each word or quasi-word and the letter, figure or word which follows it, i.e. after SIN, COS, TAN, ARCTAN, LOG, EXP, SQRT, INT, FRAC, MOD, STAND, TRIG, the group which specified any additional FUNCTION incorporated, READER, PUNCH, TELEPRINTER, JUMP, IF, UNLESS, SET, SUBR, VARY, CYCLE, REPEAT, READ, INPUT, PRINT, TITLE, LINES, SPACES, OUTPUT, CHECK, SETS, SETV, SETF, SETR, START, and nowhere else, with the exception of instructions setting Initial Values (see Chapter 11). Whenever space is used it can be replaced by $\underline{T}$ (tabulate).

20.2. Punching the Mnemonic Tape-Main Features

The mnemonic program tape should commence with a lead-in of blanks, and the instructions of the program each terminated by $\underline{L}$. The translator ignores $\underline{R}$. $\underline{L}$ is ignored if no instruction has been read, so additional $\underline{L}$'s may be punched if desired to spread out the print-up. Errors should be erased by overpunching the wrong characters with $\underline{E}$'s.

One $\underline{S}$ must be punched after each of the words and quasi-words listed in 20.1 above, except that $\underline{S}$ is optional after the last function specified in the SETF instruction. $\underline{S}$ is also optional after STOP, WAIT, LINE, EXIT and TELEPRINTER. No other $\underline{S}$'s may occur, with the following exception:

The constants on the right-hand side of an instruction setting variables to their Initial Values (see Chapter 11) may be separated by $\underline{S}$ $\underline{L}$ $\underline{T}$ or ,

Thus the number of $\underline{S}$'s in	*Is*
An arithmetic instruction	0
WAIT, STOP, LINE, EXIT or TELEPRINTER	0 or 1
JUMP IF A=B [n	2
JUMP UNLESS A=B [n	2
SETF TRIG LOG SQRT	3 or 4
Any other type of instruction, with the exception of Initial Values instructions (see above).	1

20.3. Writing and Punching-Special Points

(a) *Program Name*

The name of the program may be punched on the mnemonic tape before the setting instructions, if desired, so that it will appear at the top of the print-up. It should be preceded by

: : and terminated by L̲. It will then be output on a new line on the output writer before translation commences. If the name extends to more than one line : : must be punched at the beginning of each line.

(b) *Explanatory Notes*

Explanatory notes may be punched on the mnemonic program tape if desired, so that they will appear on the print up. It is possible to punch one short note after each instruction and symbol in the program except SETS, SETV, SETF, SETR, the [which precedes a machine block if it is on a line by itself, the) which terminates a machine code block, the R̲ or ; which terminates a TITLE instruction.

To include a note on the tape, punch : : and the note itself between the last character of the instruction proper and the L̲ which usually follows it. The translator ignores all characters between : : and L̲. There is no provision for explanatory notes which extend to more than one line.

These :'s can only be erased by overpunching.

: : may not appear before a mnemonic instruction, with the exception of the first setting instruction (see (a) above).

(c) *Machine code blocks*

These should be punched in the normal S.A.C. manner, but note that only the conventions mentioned in section 19 may be used, that zero instructions and addresses must be punched as 00 and 0 respectively, and that the zero word should be 000 : 000 L̲ or +0 L̲.

(d) *Mnemonic Program Punched in Parts*

If it is desired to punch a mnemonic program on more than one length of tape, the end of each length except the last should be indicated by S̲ L̲ immediately after the L̲ which follows a mnemonic instruction.

21. PREPARATION OF DATA TAPES

21.1. All Numbers

(a) Negative numbers should be preceded by a − sign.

(b) A + sign before positive numbers is optional.

(c) The end of a number is signified by S̲, L̲ or by T̲. No account is taken of S̲, L̲ or T̲ unless a decimal digit or point has been read, so additional page layout characters may be punched to provide a reasonable print format.

21.2. Floating-point Numbers

(a) May be punched in normal decimal form, with or without a decimal point, thus:

12.345 S̲ 142 L̲
372.0 L̲

Any number of digits up to 12 may be punched. If 12 digits are punched, their 'integer magnitude' must be less than 274 877 906 944.

(b) The argument, that is the form in (a) above, may be followed by / or $_{10}$ and a decimal exponent p, say, implying multiplication by 10^p. Thus 12.345 can be represented by: 12.345 L̲ or 12345/−3 L̲ or .12345/2 L̲ etc. The / can be replaced by a $_{10}$. The autocode allows . and $_{10}$ or / to be consecutive characters.

(c) To represent the maximum possible positive number, punch & L̲. (It is not permitted to punch − & L̲.)

(d) If a floating-point number is written with an exponent, e.g. $a_{10}b$ or a/b, then the exponent must *not* be omitted even if it is zero. The omission of an exponent is not always detected by the read routine but it leads to errors.

21.3. Integers

Integers must be punched straightforwardly, without a decimal point, and must be of less magnitude than 274 877 906 944. The character * may be used as a termination symbol instead of $\underline{S}$, $\underline{L}$ or $\underline{T}$.

To represent the maximum possible integer punch & $\underline{L}$. (It is not permitted to punch – & $\underline{L}$.)

21.4. Labels

Tapes may be labelled, that is to say they may have identifying information punched on them which does not constitute part of the data to be dealt with by the computer.

If while a READ instruction is being obeyed, an = sign is found, all following characters up to the next $\underline{R}$ or ; will be copied on to the output tape, but otherwise ignored. After the $\underline{R}$ or ; has been copied, the computer will recommence the READ action; any character read before the label was reached having been erased by the = .

The label facility may be used to cause explanatory notes to be printed on the output, but this method is inferior to the use of TITLE described in Chapter 16.1 unless the notes vary from one batch of data to another.

21.5. Triggers and Stops

The data tape may be empowered to control the program by punching triggers and stops on it.

If the data tape is so arranged that a trigger, that is, a group of characters of the form n (where n is a positive integer, is encountered while a READ A or READ I instruction is being obeyed, the computer will jump to the instruction with reference number n. The READ A or READ I instruction will not actually be obeyed: that is, the value of A or I will remain unchanged.

If the data tape is so arranged that the character) is encountered while a READ A or READ I instruction is being obeyed, it has the effect of stopping the program (see Chapter 21.8).

21.6. Error Erasure

The character ? may be used to erase incorrectly punched data. It has the effect of cancelling all decimal digits of the number being punched, together with any minus sign, decimal point, or stroke. It does not cancel a wrong character (see 21.8) nor * & = ( ) $\underline{S}$ $\underline{L}$ or $\underline{T}$.

21.7. Wrong Characters and Overflow

The effects of wrong characters and of overflow while data is being read are given in Chapter 25.5.

21.8. Summary of Effects of Characters

The following table summarises the effects of all characters, encountered while a READ instruction is being obeyed.

Characters	*Meaning or effect*
$\underline{R}$ (blank)	Ignored unless it occurs after an = , in which case it terminates the label.
$\underline{L}$	End of a number.
$\underline{T}$	
(	Jump to instruction with reference number n specified by the number before the (.
)	Causes a data wait, DTWAIT is output on the typewriter, change the value of the N2 buttons from odd to even or vice-versa to continue reading.
,	Wrong.
£	
:	
&	Maximum possible number.
*	End of an integer

/	Precedes exponent.
0 - 9	Digits.
10	Wrong.
11	
=	Precedes a label, the following characters are copied until either ; or bl is read.
+	Precedes positive numbers (optional).
—	Precedes negative numbers.
.	Appears in floating-point numbers, wrong in integers.
;	End of label.
A - Z	Wrong.
|	
$\underline{S}$	End of number.
[	Wrong.
]	
$_{10}$	Precedes exponent.
<	Wrong.
>	
↑	
~	
%	
?	Causes all characters read so far to be ignored, the read routine prepares to read a new number in the same mode.
a–z	Wrong.
— (underline)	
$\underline{E}$	Ignored.

All other codes and unallocated codes cause error stops, when the read routine detects a wrong character it stops reading and outputs 'READ ERROR' on the typewriter. The read routine is re-entered if the sign of the word generator is changed.

22. STORAGE LIMITATIONS

22.1. General

When a program is being translated and run Load and Go the R.A.P., the Autocode Translator and the Functions and Ancillaries are always in the store, along with that program. There must also be room for each of the variables and constants which are required for the calculation. The capacity of the store of the computer is limited, and this imposes limitations by which the lengths of program and/or amounts of data which can be handled at one time are restricted.

If a program is not being translated and run by the Load and Go method, however, these limitations are not so severe, for the following reasons. When such a program is being translated the R.A.P., the Autocode Translator only and certain data relating to the program must be held in the computer's store. During the running stage, the R.A.P. and the translated program must be held in the store along with the necessary Functions and Ancillaries, and there must also be room for each of the variables and constants which are required.

22.2. Applicability of 503 Autocode with Reference to Store Size

The capacity of the computer's store is measured in words. Each word occupies a location, the locations in the store being numbered from 0 upwards. The store size of the 503 is 8192 locations, but locations 7936 – 8191 hold the R.A.P. and the Interrupts (see 1.2.1 and 2.2.1) and a further 50 are used by the R.A.P. workspace and its interrupts.

22.3. Translating for a Limited Part of the Store

By means of this facility it is possible to specify a limited part of the store in which the program is to be run.

Control Tape

If it is intended to make use of this facility a control tape must be prepared. This tape specifies the lower limit of the store in which the program is to be run.

This control tape should be punched thus:

Either at the leading end of the mnemonic program tape (or the first length of the mnemonic program tape) and separated from the program by some blanks,

```
+X
+Y
*
blanks
SETV
.
.
.
```

Or on a separate tape with a closed bracket immediately after the last L̲,

```
+X
+Y
)
```

where, in both cases,

(i) X is the required lower limit, $X \geqslant 17$

(ii) $Y = 0$ if tape 2 is not to be copied onto the translated program (see Chapter 23.2).
$Y = 1$ if tape 2 is to be copied onto the translated program (see Chapter 23.2).

Both X and Y must always be set.

22.4. Capacity Calculation

To ascertain whether a long program or one which uses a lot of data at one time can be run, one may either (a) Endeavour to translate it and observe the effect achieved. See Chapters 24, 25.1, 25.2(c) and 25.3(c).

or (b) Add together the number of locations which the program's instructions, constants, variables, functions, etc., will occupy in the store (see (i) below) and compare this sum with the number of locations allocated to the program in the computer store. (See (ii) overleaf.)
N.B. This method is only approximate.
(i) The following are the numbers to be added together:

For	*Locations*
Each mnemonic instruction, an average of	2
Each different variable specified in SETS AND SETV	1
Each different floating-point constant and each different integer constant	1
If the largest reference number is n	$\frac{n}{2}$
(But if trace facility is required)	$\frac{7n}{2}$

Functions specified in SETF
(if Load and Go method is not being used):

TRIG	58
ARCTAN	39
EXP	57
LOG	40
SQRT	29
INT	41
FRAC	22
READER/PUNCH/TELEPRINTER	24
Each word in machine code block	1
Every 5 characters specified in a TITLE instruction	1

Note. It must again be emphasised that when the Load and Go method is being used the numbers given against SETF functions above should be ignored.

(ii) The number of locations which a program may occupy in the computer store is determined by whether or not the program is being run using Load and Go.

The following table lists the appropriate numbers to be compared with the sum calculated from (b) (i).

Method of Use	Compile Time	Run Time
Load and Go	4686	4686
Translate and Run Separately	4583	7220

'Compile Time' means the room available for the reference table and constants during translation.

Note. The numbers given against Translate and Run Separately will of course be altered if it is intended to Translate onto Tape for a Limited Part of the Store.

23. OPERATING INSTRUCTIONS

It is assumed in this and subsequent sections that the reader knows how to operate the computer. Except where otherwise indicated, every reference from here on to reader or punch is a reference to Reader 1 and Punch 1.

23.1. Load and Go

	Action	*Result*
Stage 1	Place Tape 1, the 503 Autocode Translator in the tape reader; type **IN**.	The tape is read in at speed. When the sum-check is found to be correct AUTO 1 UNCHEK is displayed. Control returns to the R.A.P.
Stage 2	Place Tape 2, the 503 Functions and Ancillaries Tape, in the tape reader; type **IN**.	The tape is read in at speed. When the sum-check is found to be correct AUTO2 UNCHEK is displayed and control returns to the R.A.P. If ERRSUM is displayed go back to Stage 1.

Stage 3 (translation)	Place the mnemonic tape in the tape reader, set F2=40 if TRACE facility is required, depress the B-digit if CHECKS are to be translated and type **7.**	The tape is read in and translated into the store.
	If the program is punched in parts (see Chapter 20.3(d)).	The computer will stop when the $\underline{S}$ $\underline{L}$ is reached and ERROR will be displayed.
	Place the next length of tape(s) if any in the reader and either **change the sign of the word generator or depress the MESSAGE button and type 17.**	The computer will now read in the length of tape.
	The above sequence should be followed whenever the computer reaches the end of a length of tape, with the exception of the last length.	

N.B. Re-entry by typing 17. cannot be used either in the middle of a machine code block or after a START instruction has been translated unless using the MODIFY facility. See Chapter 13.2.

	Action	*Result*
		Translation is complete when the limits of the program are displayed and control returns to the R.A.P. (see Chapter 24).
Stage 4 (running)	Place the data tape if any in the appropriate reader, set F2=40 if TRACE facility is required, depress the B-digit if CHECKS are required and type **16.** having	
either	Ensured that the N2 keys are clear	Calculation will start at the instruction specified in the START instruction.
or	set N2=n	Calculation will start at the instruction with reference number n.
and *either*	Ensured that the N1 keys are clear	Calculation will stop at the point in the program specified by the STOP instruction and END is displayed.
or	set N1=m	Calculation will stop at the instruction with reference number m and STOP is displayed.
	Each time a WAIT instruction is obeyed the program will stop,	WAIT is displayed.
	Change the setting of the N2 buttons from odd to even or vice versa.	The program will continue.
	If any further programs remain to be processed go back to Stage 3.	

23.2. Translation Stage Only

The following describes the operating procedure necessary at the translation stage if the Load and Go facility is not being used.

(a) If it is *not* intended to translate onto tape for a limited part of the store (see Chapter 22.3), proceed thus:

	Action	*Result*
Stage 1	Place Tape 1, the 503 Autocode Translator in the tape reader ; type **IN**.	As for Stage 1 of 23.1.
Stage 2 (translation)	As for Stage 3 of 23.1 except that **5.** is typed to read in the tape and not 7.	As for Stage 3 of 23.1.

(b) If it is *not* intended to translate onto tape for a limited part of the store but tape 2 is to be copied onto the output tape, proceed thus:

	Action	*Result*
Stage 1	As for Stage 1 of (a).	As for Stage 1 of (a).
Stage 2 (translation)	As for Stage 3 of 23.1 except that **5 ; S.** is typed to read in the tape and not 7. Do *not* run out any blank tape after translation is complete.	As for Stage 3 of 23.1 except that*when translation is complete the machine cycles in a keyboard loop. *1 P6
Stage 3	Place Tape 2, the 503 Functions and Ancillaries Tape, in the tape reader and **change the sign of the word generator.**	Tape 2 will be copied from the tape until the trailing blanks are reached.
Stage 4	Run out some blanks, tear off the tape, place the leading end in the tape reader and **change the sign of the word generator.**	The tape is read in at speed and sum-checked. ERRSUM is displayed if sum-check fails. P6 Repeat from b Stage 2. Otherwise END is output.

(c) If translating onto tape for a limited part of the store and if the control tape is punched on the beginning of the mnemonic program tape, proceed thus:

	Action	*Result*
Stage 1	As for Stage 1 of (a).	As for Stage 1 of (a).
Stage 2	As for Stage 3 of 23.1 except that **6.** is typed to read in the tape and not 7.	As for Stage 3 of 23.1.

(d) If Translating onto Tape for a Limited Part of the Store, and if the control tape is separate from the mnemonic tape proceed thus:

	Action	*Result*
Stage 1	As for Stage 1 of (a).	As for Stage 1 of (a).
Stage 2	Place the control tape in the tape reader and type **6.**	The control tape is read in and END is displayed when the terminating) has been read.
Stage 3	As for Stage 3 of 23.1 except that **6 ; S.** is typed to read in the mnemonic tape and not 7.	As for Stage 3 of 23.1.

(e) If it is intended to copy tape 2 on to the output tape then when translation is complete do not run out blanks but proceed as for (b) Stage 3.

(f) It should be noted that it is not necessary to continually re-read Tape 1 into the computer if it is desired to translate one mnemonic program after another. *On the other hand*, if it is desired to change from translating onto Tape to Translating into store, or vice versa, the Translator must be re-read into the computer. If this is not done × will be displayed and control will return to the R.A.P.

23.3. Running Stage Only

The following describes the operating procedure necessary at the running stage if the Load and Go facility is not being used.

	Action	*Result*
Stage 1	Place the translated program in the tape reader and type **IN**.	The tape is read in at speed and control returns to the R.A.P. If ERRSUM is displayed repeat this stage.
Stage 2	Place Tape 2, the 503 Functions and Ancillaries Tape in the tape reader and type **IN**.	The tape is read in at speed and control returns to the R.A.P. If ERRSUM is displayed repeat from Stage 1.
Stage 3	As for Stage 4 of 23.1.	As for Stage 4 of 23.1.

If a combined tape (the translated program and tape 2) was produced as in 23.2(b) proceed as above but omit Stage 2.

24. THE TRANSLATED PROGRAM

The translated program tape is punched in a special relocatable binary code suitable for input under the R.A.P.

The limits of the program are displayed when translation is complete. These are of the form M – P and Q – R and indicate that locations M to P will be used for the main program, locations Q to R for constants; reference table and workspace. If the program is translated onto tape M = 16.

If an overlong program is translated to its end, as suggested in Chapter 25.2(c), P will not be less than Q. The excess is, in fact (P–Q+1) words.

The R.A.P. pointers which indicate the first and last free locations are updated after the program and tape 2 have been read into the store. When translating Load and Go the necessary instructions to update the R.A.P. pointers are carried out by the translator after the START instruction has been compiled.

The two modes of translation (onto tape and into store) are not compatible, i.e. it is not possible to change from translating onto tape to translating into store without re-inputting the translator. If a change of mode is attempted without first re-inputting the translator then × is typed on the output writer and control is transferred to the R.A.P. where it awaits a message.

25. ERROR INDICATIONS AND ACTION TO BE TAKEN

25.1. Summary of messages output during translation

Message	*Meaning*	*Further Action*
ERRSUM	Tape sum-check has failed on input and control is transferred to the R.A.P.	Re-input the tape.
END	The current stage of operation is complete.	Proceed with the next stage of the operation.
*ERROR	An error in the setting instructions.	Remove the tape, correct the setting instructions and re-commence translation.
ERROR	An error in an instruction or a control tape or $\underline{S}$ $\underline{L}$ i.e. stop at the beginning of a line. In the latter case the error is ignorable.	Mark the tape, change the sign of the word generator to continue translating and set F2=04 if it is desired to suppress further output. If the error occurred in a control tape, correct the tape and re-enter by typing 6. (see also Chapters 25.3 and 25.4).
×	An attempt to change the mode of translation.	Re-input the translator.
NOROOM	The program is too long for the store.	Continue as for ERROR.

25.2. Errors during Translation

The error stops described below are those which may occur while the operations specified in Chapters 23.1 and 23.2 are being carried out.

The Translator will stop

(a) If there is an obvious error in the setting instructions.

(b) If there is an obvious error in any other part of the mnemonic program.

(c) If the translated program is becoming so long that there will not be room for it and its data in the computer's store at the running stage. In this case NOROOM is displayed.

Examples of (a)

(i) Word or quasi-word spelt wrongly.

(ii) Unrecognised word used.

(iii) Same letter set twice.

(iv) Omitting SETR.

Examples of (b)

(i) Using a letter or FUNCTION which has not been set (N.B. Use of A8 where only A(7) has been set, and use of AI when I < 0 will not be detected by the Translator, and errors will arise).

(ii) Using a constant with excessive 'integer magnitude'.

(iii) Using integers and floating-point quantities in wrong combinations.
Using an integer variable on either side of a FUNCTION instruction not applicable to integers is not detected and will lead to errors at run time.

(iv) Using an integer variable in an instruction to divide.

(v) Using a constant on the left-hand side of any instruction.

(vi) Word or quasi-word spelt wrongly, or unrecognised word used.

(vii) Giving an instruction a reference number higher than the maximum specified in the SETR instruction, or giving the same reference number twice.

25.3. Action to be Taken on an Error Stop During Translation

(a) If the translator stops because of an error in the setting instructions the tape should be removed, a corrected tape punched, and translation recommenced from the beginning. Use may be made of the method described in Chapters 20.3(d) and 23.1 Stage 3, but the whole of the setting intructions must be repunched.

(b) If the translator stops because of an obvious error in the body of the program, the following courses are open:

(i) Take the tape away and correct it. This is not very profitable, as there may be more errors on it.

or (ii) Mark the tape at the point at which the error was found. Set up 04 on the F2 buttons of the keyboard and change the sign of the word generator, that is, change the setting on the F1 buttons from 40 (negative) to 00 (positive), or vice-versa. The translator will 'read over' the rest of the mnemonic tape and check it for obvious errors but no further output will take place. It will stop each time it finds an error, whereupon the tape should be marked, and checking recommenced by again changing the sign of the word generator. Then take the tape away and correct all errors.

or (iii) Mark the tape at the point at which the error occurred and remove it from the tape reader. Punch a correction tape consisting of:
blanks
the correct version of the instruction $\underline{\text{L}}$
$\underline{\text{S}}\ \underline{\text{L}}$
$\overline{\text{blanks}}$
Place this in the tape reader, and cause it to be read by changing the sign of the keyboard word (see (ii) above). Then put the main mnemonic tape back in the tape reader, with the first character of the next instruction in the reading position, and again change the sign of the word generator.

(c) If the translated program is becoming so long that there will not be room for it and its data in the computer's store, NOROOM is displayed. If it is desired to continue translation, in order to determine the exact amount by which store capacity is exceeded, change the sign of the word generator. See Chapter 25.1.

25.4. Fast Method of Detecting Errors on the Mnemonic Tape

If the translation process is being carried out separately, quite a lot of time can be wasted in attempting to translate mnemonic tapes in which there are errors. Some users avoid this possibility by always making the translator 'read over' a mnemonic tape before actually attempting to translate it. This is done by operating as in Chapter 23.2 but with the F2 buttons set at 04. No output takes place, but the translator rapidly 'reads over' the mnemonic tape, stopping if any obvious error is found. The actions taken on a stop are similar to those stated in Chapters 25.3(a) and 25.3(b) (ii).

25.5. Summary of Messages Output at run time

Printed on output writer	*Meaning*
C	is displayed when entering at 16. if there is any non-zero setting on the word generator (apart from the sign bit which is ignored in this context).
N	is displayed when entering at 16. if the word generator is zero (apart from the sign bit).
STOP	is displayed when the selected stop facility is being used to indicate that the instruction with the specified reference number has been reached.

END	is displayed when the mnemonic instruction STOP is obeyed and control is transferred to the R.A.P.
Printed on output writer	*Further Action*
WAIT	Change the value of the N2 keys from odd to even or vice-versa and the program continues.
DTWAIT	Change the N2 keys as for WAIT and the next number on the data tape is read.
READ 0 / PUNCH 0	Type a message m. where $0 \leqslant m \leqslant 7$ See Chapter 17.3.

Error indications at run time

Change the sign of the word generator to continue for group (a). Continuation is not possible for group (b).

Message	*Meaning*	*Effect*
Group (a).		
READ ERROR	Wrong character on data tape, or number read has excessive 'integer magnitude', or a '.' in an integer.	A new number is read.
PRINT ERROR	The floating-point number to be printed is of non-standard form.	? 0 is printed in the specified mode.
TRIG ERROR	SIN, COS or TAN with argument too great.	Argument is set equal to max. permitted value.
LOG ERROR	LOG with argument $\leqslant 0$	Calculate LOG (|A|).
EXP ERROR	EXP with argument $> 254 \log_e 2$	Calculate EXP ($254 \log_e 2$)
INT ERROR	I = INT A with argument too great for fixed point representation.	Function continues.
SQRT ERROR	SQRT with negative argument.	Calculate SQRT (|A|).
RPUNCH ERROR	READER I/n when I/n > 7 or I/n < 0 or PUNCH I/n where I/n > 7 or I/n < 0	Set tape reader 1 or punch 1.
Group (b)		
SUBR ERROR / END	Subroutines to excessive depth.	Control is transferred to the R.A.P. to await a message.

26. AUTOCODE POST MORTEM

26.1. Function

To output the current values of variables of the Autocode program held in store (the object program), as specified by a control tape.

26.2. Method of Use

The post-mortem must be input after each program requiring its use by typing IN. The mnemonic program tape should be placed in Tape reader 2, and post mortem may then be entered by typing AUTOPM.

The post mortem first reads the setting instructions from the head of the mnemonic tape of the object program, on tape reader 2, forms its alphabet table and then waits in a keyboard loop.

When the parity of the N2 keys of the word generator is changed, the control tape is read, also from reader 2, the variables specified are output, together with any page layout characters on the control tape until either * or) is read.) will cause the program to 'WAIT', and when the parity of the N2 keys is changed the program continues reading the control tape. * causes the post mortem to exit to the R.A.P. to await a message.

26.3. Control Tape

The control tape should contain the following:

£n $(1 \leqslant n \leqslant 7)$	(optional). n specifies the number of the output device on which the values are to be printed. If n is not set then the current device set by the object program will be used, except in the case where the current device marker has value 0 in which case output will be on punch 1. £n must be terminated by $\underline{\text{L}}$, $\underline{\text{S}}$ or $\underline{\text{T}}$.

A list of the variables whose values are required, in any one of the following forms:

I = or A =	when I/A only is required
In = or An =	when In/An only is required
Im–n = or Am–n =	when Im to In or Am to An inclusive are required.

Any page layout characters may appear on the control tape. These will be copied on to the output but will otherwise be ignored.

)	If there are further variables to follow later, e.g., on another tape.
*	If the program is required to exit to the R.A.P. to await a message.

All variables are printed either in the form

PRINT A,8/ or PRINT I,12

The current output device marker is saved on entry to the post mortem and reset before exit or whenever the output device is changed during the course of the program. The output device may be changed at any time during output by setting £n on the control tape.

26.4. Errors

Message	*Meaning*	*Further Action*
NOTSET ERROR	A letter which has not been set in the object program is specified on the control tape.	Change the sign of the word generator to continue reading the control tape.
SETIN ERROR	A wrong character has been read on the mnemonic tape of the setting instructions.	Change the sign of the word generator to continue reading the setting instructions.
ALPHA ERROR	An impermissible character has been found on the alphabet (control) tape.	Change the sign of the word generator to continue reading the alphabet tape.

26.5. Re-entry

AUTOPM may be entered several times during the running of a program, but must be re-input whenever another object program is run or translated. AUTOPM must always be input last, i.e. after the translated program, otherwise it may be overwritten.

APPENDIX I

SUMMARY OF INSTRUCTIONS

Note — In the examples below,

A, B, C and D	represent	Floating-point variables.
I, J, K and L	represent	Integer variables.
l, m and n	represent	Positive integer constants.
p, q and r	represent	Any integer constants.
x, y and z	represent	Floating-point constants.

Any variable except the one before the = sign may be replaced by a constant.

ARITHMETIC
(Chapter 4)

A=B	A=–B	I=J	I=–J
A=B+C	A=–B+C	I=J+K	I=–J+K
A=B–C	A=–B–C	I=J–K	I=–J–K
A=B*C	A=–B*C	I=K*K	I=–J*K
A=B/C	A=–B/C	—	—

FUNCTION
(Chapter 5)

A=SIN B	
A=COS B	
A=TAN B	
A=ARCTAN B	
A=LOG B	
A=EXP B	
A=SQRT B	
A=INT B	I=INT A
A=FRAC B	I=MOD J
A=MOD B	
A=STAND I	

SETTING AND START
(Chapter 7)

SETS	(Integer variables)
SETV	(Floating-point variables)

SETTING AND START
(Chapter 7)

SETF	(Functions and peripheral equipment)
SETR n	(Maximum reference number)
START m	(Starting reference number)
In SETF	(i) TR!G covers SIN, COS and TAN
	(ii) MOD and STAND need not be mentioned.

JUMP
(Chapter 8)

```
JUMP [ K
JUMP IF A=B [ K             JUMP IF I=J [ K
JUMP UNLESS A=B [ K         JUMP UNLESS I=J [ I
```

(K may have a numerical suffix)

Any permitted arithmetical instruction or function instruction may be substituted for A=B or I=J, and may have > or < in place of =

OTHER CONTROLS		
(Chapters 9 and 10)	SUBR n	SUBR I
	EXIT	
	STOP	
	WAIT	
	(I may have a numerical suffix)	
INITIAL VALUES		
(Chapter 11)	SET A(n:m) = x, y, z, ...	
	SET I(n:m) = p, q, r, ...	
VARY AND CYCLE		
(Chapter 12)	VARY A=B:C:L	VARY I=J:K:L
	CYCLE A=B:C:D	CYCLE I=J:K:L
	CYCLE A=x, y, z, ...	CYCLE I=p, q, r, ...
	REPEAT A	REPEAT I
	(B, C, D, J, K, and L may have simple suffices only)	
INPUT		
(Chapter 14)	READ A	READ I
		INPUT I
	In Input I, I may have a simple suffix only.	
OUTPUT		
(Chapters 15 and 16)	PRINT A, n:m	PRINT A, I:J
	PRINT A,n	PRINT A,I
	PRINT A,n/	PRINT A,I/
	PRINT A	
	PRINT I,n	PRINT I,J
	PRINT I	
	OUTPUT I	
	LINE	
	LINES I	
	SPACES I	
	TITLE	
DOUBLE PAPER TAPE STATION		
(Chapter 17)	READ I/n	PUNCH I/n
		TELEPRINTER
CHECK		
(Chapter 18)	CHECK A	CHECK I

APPENDIX 2

SUMMARY OF OPERATING INSTRUCTIONS AND ERROR INDICATIONS

A. LOAD AND GO

1. Operation

(a) To read in appropriate Tape 1, type **IN.**
Input is sum-checked.

(b) To read in appropriate Tape 2, type **IN.**
Input is sum-checked.

(c) To read in mnemonic tape, type **7.** with:
 B = 1 to translate CHECK'S
 F2 = 40 to Incorporate TRACE facility.

(d) If the mnemonic program is punched in parts, to recommence translation either **change the sign of the word generator or type 17.**

(e) To enter the program at the point specified in the START instructions, type **16.** with keyboard N2 = 0. To enter at the instruction with reference number n, type **16.** with keyboard N2 = n. To exit from the program at the point specified in the STOP instruction set N1 = 0. To exit at the instruction with reference number m set N1 = m. Keep the keyboard set with:
 B = 1 for CHECK'S to be obeyed.
 F2 = 40 for TRACE facility to be used.

(f) If there are any WAIT instructions in the program, change N2 from odd to even, or vice versa, when ready for calculation to continue.

2. Error Stops during Translation and Action to be taken

If the translator stops without completing the translation, including the output of the details of storage space required by the translated program, then:

(a) If *ERROR is displayed remove the tape and correct the setting instructions.

(b) If ERROR is displayed, and the tape in the reader is not a control tape then:

Either (i) Mark the tape.
(ii) Set F2 = 04 on the keyboard.
(iii) Change the sign of the word generator, that is, change F1 from 40 to 00 or vice versa.
(iv) Repeat (i) and (iii) each time the translator stops until the end of the tape is reached. Then remove the tape, correct it at all marked points, and recommence.

or (i) Mark the tape and remove it.
(ii) Punch a correction tape comprising: blanks correct instruction $\underline{\underline{L}}$ $\underline{\underline{S}}$ $\underline{\underline{L}}$ blanks.
(iii) Change the sign of the word generator to read correction tape.
(iv) Change the sign of the word generator to continue reading main tape at beginning of next instruction.

(c) If NOROOM is displayed the program is overlong. See Chapter 25.3(c) for recommended action, and refer to Chapters 22.4 and 24.

B. TRANSLATION STAGE ONLY

1. Operation

(a) To read in appropriate Tape 1, type **IN.**
Input is sum-checked.

(b) (i) If it is not intended to translate onto tape for a limited part of the store, to translate mnemonic tape, type **5.**

(ii) If it is *not* intended to translate onto tape for a limited part of the store but tape 2 is to be copied onto the output tape, to translate the mnemonic tape type **5;S.** and when translation is complete change the sign of the word generator to copy tape 2. To check the tape produced, place the leading end in the tape reader and change the sign of the word generator.

(iii) If it is intended to translate for a limited part of the store, and if the control tape is punched on the beginning of mnemonic tape, then to translate mnemonic tape type **6.** If Y = 1 on the control tape then proceed as in (ii) above when translation is complete.

(iv) If translating for a limited part of the store and control tape is separate from mnemonic tape, to read in the control tape type **6.** To read in mnemonic tape type **6;S.** If Y = 1 on the control tape then proceed as in (ii) above when translation is complete.

In all cases enter with:
B = 1 to translate CHECKS
F2 = 40 to incorporate TRACE facility.

(c) If the mnemonic program is punched in parts, to recommence translation either **change sign of the word generator** or type **17.**

2. For a summary of Error Stops during Translation and Action to be taken, refer to Chapter 25.1.

C. RUNNING STAGE ONLY

1. Operation

(a) To read in translated tape, type **IN.**
Input is sum-checked.

(b) To read in appropriate Tape 2, type **IN.** (Omit this for 1(b)(ii) above).
If sum-check failure occurs, start again at (a).

(c) Now proceed sequentially from A.1(e).

2. For a summary of Error-Indications during Running, refer to Chapter 25.5.

APPENDIX 3

FLOATING-POINT REPRESENTATION

(a) General

Any number A can be represented in many ways by a pair of numbers (a,b) which satisfies the equality

$$A = a \times 2^{b}$$

In which a is called the mantissa or argument and b the (binary) exponent.
For example, 6 can be represented by (6, 0) or by (.75,3) or by (12,–1).

(b) Representation in 503's as used in the 503 Autocode

The representation used in the 503 is described in 1.2.1. The nine least significant digits of the word represent (b+256), not (b+255).

APPENDIX 4

ADDITION OF FURTHER FLOATING-POINT FUNCTIONS

Note: All examples in this appendix assume that it is desired to add three new functions, and that these will be specified by mnemonic instructions of the forms:

```
A=LIM B
A=MIN B,C,D
A=Norm B
```

where LIM, MIN and Norm are the names of the functions and A, B, C and D are floating-point variables.

Additional functions may be functions of one or more integers or floating-point variables, and up to eleven such functions may be added.

SUMMARY

In order to be able to use functions which are not already available in the 503 Autocode the following actions are needed:

1. Allocate a name to each such function.
2. Write additional functions subroutines in a style similar to S.A. Code, by which the functions are formed.
3. *Either* (a) Produce a tape of these subroutines, punched in the conventions of the autocode system (Additional Functions Tape), to be used in conjunction with tape 2.
   or (b) Produce a modified version of the appropriate tape 2, incorporating these additional subroutines.
4. *Either* (a) Produce a tape by means of which the Autocode Translator can be adjusted each time it is required to translate a mnemonic program using these additional functions (Additional Names and Lengths Tape).
   or (b) Produce a modified version of the appropriate tape 1 which will translate mnemonic programs using the additional functions.
5. Specify any additional functions used in a mnemonic program in the normal way, that is, by including their names in the SETF instruction, and use the additional or modified tapes correctly.
6. Ensure that modified versions of Tapes 1 and 2 do not get mixed up or used in conjunction with the standard versions, or differently modified versions.

1. ALLOCATING NAMES TO FUNCTIONS

Each name may be any combination of at least 3 of the letters of the alphabet, provided that the last 7 of them are different from the last 7 letters of any other function name in use. They may consist of any combination of upper and lower case letters.

for example: LIM MIN and Norm

2. WRITING THE REQUISITE SUBROUTINES

Each subroutine must be written as one block of S.A. code without data or label introductions, the only permitted forms of addresses are integers and relative (see 2.1.2.3), and must comply with the following specification:

(i) The entry point is the first instruction in 0,.

(ii) On entry, the first (or only) parameter is in the accumulator. Additional parameters are picked up in the following manner.
Placing the address of the m$^{th.}$ parameter in location 12 (for example) its current value in the accumulator is achieved by the instructions:

```
        00   15   /   30   m-1
        20   12   /   30   0
```

There is no limit on the number of parameters.

(iii) Exit from the function must be made by the instruction pair:

```
        00   15   /   40   n
```

where n is the number of parameters in the function, with A, the required result, in the accumulator.

(iv) The subroutines should be written and punched as a series of whole words, i.e. 2 instructions to a line. When two instructions are to be stored in the same location they must be written and punched on the same line.

3. (a) PRODUCING AN ADDITIONAL FUNCTIONS TAPE

(i) Punch S.A. Code tapes of the subroutines, complying with the methods of punching used in preparing tapes for input by S.A.P. and with the following special provisions:
The comments between round brackets are not allowed.
The first subroutine must be preceded by * and one * must occur between each subroutine and the next. The last subroutine must be followed by)
If the subroutines are punched on more than one tape, each tape except the last must end with – –
The subroutines may be punched in any sequence, but a note should be kept of the actual sequence used; say (1) LIM, (2) MIN, (3) Norm. Note: The subroutines must be punched on 5 hole tape

(ii) Place Tape 3 in the tape reader and type IN. Input is sum-checked; the output of ERRSUM indicates sum-check failure.

(iii) Place the S.A. code tape of the additional subroutines in the tape reader and type AUTO3;1. The coding routines on Tape 3 will now translate the subroutines into a form suitable for use in conjunction with Tape 2, producing what is known as an Additional Functions Tape. If the subroutines have been punched on more than one tape, then when each tape has been translated, replace it by the next in turn, and type 2. to cause translation to continue.

3. (b) PRODUCING A MODIFIED TAPE 2

(i)-(iii) Proceed exactly as in 3(a) except that the last subroutine must end with 44 7898: (instead of) . Then, without running out any blanks or tearing off the Additional Functions Tape, continue as below.

(iv) Place the leading end of a Standard Tape 2 in the tape reader, and type **4.** This will cause the relevant portions of Tape 2 to be copied, so that the whole output constitutes Autocode Tape 2 (modified). Care should be taken to mark it properly, e.g. 503 Autocode with LIM, MIN and Norm, Tape 2.

(v) To check the output tape, run out some blanks, tear off the output tape and place its leading end in tape reader 1, type 5. to cause the tape to be read in and sum-checked. If ERRSUM is displayed repeat from 3(b) (iii) otherwise END is printed and control is transferred to the R.A.P.

4. (a) ADJUSTING THE AUTOCODE TRANSLATOR

(i) Prepare an Additional Names and Lengths tape, consisting of NAME $\underline{S}$(or $\underline{T}$)n $\underline{L}$
for each function, where n is the number of locations taken up by the subroutine for the function.
The functions must be specified in the opposite sequence to that in which they were punched for coding, so in our example we would have, say,

Norm $\underline{S}$ 42 $\underline{L}$
MIN $\underline{S}$ 20 $\underline{L}$
LIM $\underline{S}$ 33 $\underline{L}$

(ii) Read the appropriate Tape 1 into the computer in the normal way.

(iii) Place the leading end of the Additional Names and Lengths tape in the tape reader, and type **1786.**
The Autocode Translator will read the first name and length on the tape, adjust itself accordingly and END will be typed. Type **1786.** repeatedly until all the names and lengths have been read and dealt with.

(iv) The Autocode Translator is now adjusted to translate mnemonic programs using the additional functions, and can be used in the normal way.

4. (b) PREPARING A MODIFIED AUTOCODE TRANSLATOR

(i)-(iii) Proceed exactly as in 4 (a) (i)-(iii), and then continue as below.

(iv) Make sure that there is plenty of tape in the punch, and type **1863.**

(v) The Autocode Translator will now punch out a suitably modified version of itself, and care should be taken to mark it properly, e.g. 503 Autocode with LIM, MIN and Norm, Tape 1.

(vi) To check the output tape, run out some blanks, tear off the output tape, place the leading end in tape reader 1 and change the sign of the word generator. If the check-sum fails, ERRSUM is displayed and the operation must be repeated from stage (iv), otherwise END is typed and control is transferred to the R.A.P.

5. USING 503 AUTOCODE WITH ADDITIONAL FUNCTIONS

The following notes amplify the provisions of the basic specification of 503 Autocode.

(a) *Writing the Mnemonic Program*
The names of any additional functions used in a mnemonic program must be specified in the SETF instruction in the normal manner and the way in which each additional function instruction is written must conform with the usual conventions.
Functions of more than one variable must obey the following further restrictions:

(i) The first parameter may be either a constant or a variable with any form of suffix. The second and subsequent parameters must be variables with at most numerical suffices.

(ii) The parameters must be separated by commas and the last variable must be followed by $\underline{L}$ or ::

(iii) A comma following the last variable will cause an error stop on translation. A function call with the wrong number of parameters will not however be detected and will lead to run-time errors.

(b) *Load and Go Operation*

(i) Using a modified form of Tape 1 :
Read the modified Tape 1 into store. Read in either a modified form of Tape 2 or an additional functions tape followed by the standard form of Tape 2. Then type **7.** to translate in the normal way. Type **16.** to run the program.

(ii) Using the standard form of Tape 1 :
Read the standard version of Tape 1 into store, then modify it by the method described in

paragraph 4(a). The autocode translator must be adjusted as in paragraph 4(a) before inputting either a modified form of Tape 2 or an additional functions tape followed by the standard form of Tape 2. Then type **7.** to translate in the normal way. Type **16.** to run the program.

Restriction: Additional functions must not be added when using the Load and Go facility after entering Tape 1 to translate a program, without first re-inputting Tape 1.

(c) *Translate and Run Separately*

(i) Translating the Mnemonic Program.
Either use a modified form of Tape 1, produced as described in 4(b) above, or use the standard form, modifying the translator after input by the method described in 4(a).

(ii) Running the Translated Program.
If a modified form of Tape 2 has been punched, as described in 3(b) above, this should be used instead of the standard form, operating in the usual way.
If only an Additional Functions Tape has been produced, this should be read in (by typing IN.) after the translated program but before the standard Tape 2. Otherwise, operate in the usual way.

6. INCOMPATIBILITY OF STANDARD AND MODIFIED VERSIONS OF THE TAPES

A mnemonic program which does not in fact use any additional functions may, if desired, be translated by a modified version of the translator, but the translated program tape so produced can only be run under a correspondingly modified version of Tape 2.

Care should be taken to ensure that each modified tape is kept with its pair, and does not get mixed up with standard tapes or with tapes modified for other sets of additional functions.

APPENDIX 5

DETAILS OF FLOATING-POINT SUBROUTINES

This appendix gives sufficient details of the READ and PRINT subroutines of Tape 2 of 503 Autocode to enable users to employ them in machine code blocks (see Chapter 19) and additional function subroutines (see Appendix 4).

1. DETAILS OF READ AND PRINT SUBROUTINES IN 503 AUTOCODE

Throughout this section, all quantities except I and J are standard floating-point numbers unless otherwise stated.

READ

To read a floating-point number from the input tape, and place it in the accumulator, enter

```
          73       15       :       40       7679
```

To read an integer from the input tape and place it in the accumulator, enter

```
          73       15       :       40       7673
```

For details, see Chapters 14 and 21.

PRINT

For Format Corresponding to	*Set Parameter*	*In Location*	*Place A or I in the Accumulator and Enter*
PRINT A,m:n PRINT A,I:J	20 m : 00 m+n 20 I : 00 I +J	7323	73 7620 : 40 7325
PRINT A,m PRINT A,I	10 m : 00 m 10 I : 00 I	7323	73 7620 : 40 7325
PRINT A,m/ PRINT A,I/	40 0 : 00 m 40 0 : 00 I	7323	73 7620 : 40 7325
PRINT A	—	—	73 7620 : 40 7325
PRINT I,m PRINT I,J	+m +J	7324	73 7620 : 40 7326
PRINT I	—	—	73 7620 : 40 7326

For details, see Chapter 15.

Modifications to the Print Routine

(a) To output + instead of $\underline{S}$, modify the contents of location 7577 from +64 to +29.

(b) To cause the sign of the number to be output before the suppressed leading zeros, modify the contents of location 7501 from 30 7605 : 10 7530 to 40 7513:

(c) To cause leading zeros to be output instead of spaces and to be preceded by the sign, modify the contents of location 7499 from 44 7493 : 02 7582 to 44 7493 : 30 7569 and 7507 as in (b) above.

It is also possible to modify location 7499 so that certain other characters are output instead of leading zeros.

APPENDIX 6

EXAMPLE OF USE OF THE ELLIOTT AUTOCODE PROGRAM SHEET

The accompanying sheet is an example of the method of writing programs in autocode employed at Elliott Brothers (London) Ltd.

WRITING ON THE FORM

In general, the REF. column is used for reference numbers, while instructions and remarks are written in their appropriate columns. The use of the FLOW column is at the discretion of the programmer: one possible method of indicating jumps, etc., is shown.

Working through the example, the following special cases will be observed:

(a) REF. column used for :: before program name, and the program name is written across the page (Chapter 20.3(a)).

(b) An additional Ⓛ must be directly specified: the punch operator is instructed to ignore lines left completely blank. (Chapter 20.2.)

(c) Where any instruction is too long to be written on one line, a large X appears in the END column, and instruction is continued on the next line.

(d) TITLE instructions are written out in full. (Chapter 16.1.)

(e) :: is written in the END column where a remark is to be punched. (Chapter 20.3(b)). (If a remark is not to be punched, the END column is left blank.)

PUNCHING FROM THE FORM

General

(a) Lead in with blanks followed by Ⓛ

(b) Punch whatever is written in the REF. and INSTRUCTION columns on one line, then punch Ⓛ and go on to the next line, and so on.

(c) Finish off with Ⓛ and run out on blanks.

(d) Punch one Ⓢ after the 'words'
ARCTAN, CHECK, COS, CYCLE, EXP, FRAC, IF, INPUT, INT, JUMP, LINES, LOG, MOD, OUTPUT, PRINT, PUNCH, READ, READER, REPEAT, SET, SETF, SETR, SETS, SETV, SIN, SPACES, SQRT, STAND, START, SUBR, TAN, TELEPRINTER, TITLE, TRIG, UNLESS, VARY, and nowhere else.

(e) Erase Ⓔ may occur anywhere on the program tape.

Exceptional Cases

(f) Where :: appears in the REF. column, punch :: and then copy everything else on that line, with spaces between words, and end with Ⓛ

(g) Where :: appears in the END column, punch :: and then copy the REMARKS column, with spaces between words, before punching Ⓛ

(h) Where there is no :: in the END column, do not copy the remarks.

(i) Where there is X in the END column, or at the end of the line, go on to the next line without punching Ⓛ

(j) Take no notice of the FLOW column entries.

(k) If you see Ⓛ written on a line, just punch Ⓛ once and go on to the next line.

(l) After TITLE punch Ⓢ followed by all the characters in the circle followed by Ⓛ

(m) An entry like Ⓢ⑥ means six sp's

Scientific Computing Division.....ELLIOTT BROTHERS (LONDON) LTD.

2.1.4 EXAMPLE

503 AUTOCODE PROGRAM

Sheet 1 of N

FLOW	REF.	INSTRUCTIONS	END	FLOW	REMARKS
	∷	EXAMPLE OF USE OF AUTOCODE PROGRAM SHEET			
		Ⓛ			
		SETS I(64) J(64) K(64) L(5)	X		
		M(10) NP(2)			
		SETV A(10) BC			
		SETF PUNCH READER TRIG			
		SETR 30			
	17)	PUNCH 0	∷		TO SUPPRESS PRINTING TYPE 0 ON DEMAND
		TITLE Ⓛ$\textcircled{56}$ SPECIFY Ⓢ CHARACTERS Ⓢ			
		REQUIRED Ⓢ THEN Ⓢ ENCIRCLE Ⓢ			
		WHOLE Ⓢ TITLE Ⓢ AS Ⓢ SHOWN;			
⑰		SET L(0:5)=0,1 Ⓢ 2,3 Ⓣ 4,5			
		READ N			
		P=N+12			
		SUBR 23			
		CYCLE K=N:4:P			
		CYCLE A=1:·5:20			
		B=STAND K			
		A1=SIN A			
		A2=COS B			
		PUNCH 2			
		PRINT A1,8/			
		PRINT A2,2:6			
	7)	REPEAT A			
From Sheet 5	8)	REPEAT K			
		VARY I=1:1:20			
		K=I*I	∷		FORM I^2+I IN JI
		JI=K+I			
		REPEAT I			
⑦		@30 L:501			
		00I/20J			
		)			

RM/A 203

APPENDIX 7 : LINEPRINTER FUNCTIONS AND 5-CHANNEL INPUT

GENERAL

The version MOD/LP of 503 Autocode uses the lineprinter as an output device, as well as punched tape and the typewriter, and accepts either 5-channel or 8-channel Elliott-code data tapes. There are three extra functions in this version: LPSET, LPSKIP and LPCOL. They are used as normal functions, with an integer variable on the left hand side and an integer variable or constant as the parameter on the right hand side of the instruction.

The operation of the lineprinter differs from tape output in that the characters are stored in a buffer and the whole line is output at one time. The length of the buffer is 121 locations, of which the first location contains the vertical format control character. When the lineprinter has been set as the current output device, all characters to be output using the instructions PRINT,OUTPUT,TITLE,LINE etc. are stored in the buffer in the correct order. The contents of the buffer are printed either when it becomes full or when a "newline" character is to be output. Thus, provided that the line is not more than 120 characters long, the printed results are consistent with those produced from punched tape. Tabulating may be used, and "top of form" may be given.

THE INSTRUCTION I=LPSET J

This performs certain initial operations on the system. It is used only once in the program, at the beginning and before print and read statements. 'I' can be any of the programs integer variables whose value may be lost at the given point. The parameter J is an integer variable or constant specifying the vertical format.

The following operations are performed by LPSET:

- prepares for lineprinter output and the option of 5-channel input,
- sets the lineprinter as the current output device,
- clears the lineprinter buffer and sets tabs for every sixth location,
- the count which indicates how many characters have been stored into the buffer is set to zero, i.e. the first character to be output is to be stored in the first location of the buffer,
- the value of the variable I is set to the absolute address of the beginning of the buffer, (which may be used in a machine-code block),
- the first location of the buffer is set to the value of J.

The content, p, of the first location of the buffer specifies where the line is to be printed:

$0 \leq p \leq 30$	Move p + 1 lines and print
p = 31 or 32	Overprint, i.e. no line feed
$33 \leq p \leq 62$	Look for configuration (p-32) on the paper tape control loop, and print.
p=63	If there is a control loop, find the "top of form", i.e. channel 8, and print, otherwise move one line, and print.

Note, that the actual printing doesn't take place before the "newline" character terminating the line is given.

If the value of p is greater than 63, or is negative, it is given the value 63. In practice, p will usually have the values 0 or 63. Usually one gives I = LPSET 63 which causes the first line to be printed at the top of a new page. When the buffer is printed, the first location is automatically cleared, so that output can be on consecutive lines.

The following instructions, LPSKIP and LPCOL, cannot be given before the LPSET instruction.

THE INSTRUCTION I=LPSKIP J

This alters the value of p. The variables I and J are treated in the same way as above, but if J is negative the value of p is unchanged. Thus the instruction I=LPSKIP 63 moves the paper to the beginning of a new page.

THE INSTRUCTION I=LPCOL J

This changes the value of the written buffer-length count. I and J are integer variables (J may also be a constant). If $1 \leq J \leq 120$, the count takes the value of J, and characters to be output are stored in locations J onwards of the buffer. If J=0 or is greater than 120, the count is given the value 120, if J is negative the count is unchanged. The value of I becomes the previous value of the count.

METHOD OF USE

The functions LPSET, LPSKIP and LPCOL must appear in the setting instructions SETF. When the lineprinter is the current output device, all printing instructions, including data tape headings to be copied, cause output to the lineprinter. When a new line character is to be output the line is printed and character storing begins from the beginning of the buffer again, unless an LPCOL instruction specifies otherwise.

If the buffer exceeds 120 characters, the line is automatically continued on the following line.

If LPSET has been obeyed and the output device has been changed by a PUNCH instruction, the instruction PUNCH 4 resets the lineprinter as the current output device.

One extra LINE should be given at the end of the program to ensure that the last line is printed.

All lower case letters are printed as upper case letters. The character Ⓡ (runout) and Ⓔ (erase) are ignored.

STORAGE REQUIREMENTS

The functions occupy the following space:

LPSET	251 locations
LPSKIP	12 locations
LPCOL	12 locations
total :	275 locations

N.B. The translator AUTO 1 MOD/LP is not able to test whether I and J are integers.

ERROR INDICATIONS

The instruction LPSET tests whether the right version of tape 2 is being used. If not, RAP displays the message ISSUE ERROR. If it is attempted to obey LPSKIP or LPCOL before LPSET has been obeyed, the program comes to a stop.

Before a line is printed, the lineprinter control word is tested. If it is zero, the line is printed, otherwise one of the following messages is displayed:

MESSAGE		CONTROL WORD				
	bit	5	4	3	2	1
LPERRU unavailable		0	0	0	0	1
LPERRT throat		0	0	0	1	0
LPERRF functional		1	0	0	0	0
LPERRS summed		0	0	0	1	1
LPERRE error		1	0	0	0	1
LPERRD functional		1	0	0	1	0
LPERRC combined		1	0	0	1	1
(LPERRV vacant		0	0	0	0	0)

The message in brackets should not occur.

After displaying LPERRT the program continues immediately. After any other error message the program continues automatically after the error has been corrected by the operator and the lineprinter has been put into the auto state again.

The lineprinter functions do not check the "busy" state of the lineprinter. Thus, if the lineprinter goes into an error state during a paper skip (top of form etc.), then ERRINT4 may occur. The operator may then continue by typing

```
CONT;ERRINT.
```

<u>5-CHANNEL DATA INPUT</u>

LPSET prepares for either 5- or 8-channel data tapes to be used. The system automatically tests the state of the MODE button, and if depressed, every 5-channel character read is translated to the corresponding 8-channel character. (Except when the characters are read by a machine code block.)

For example, the value of the character read by the instruction INPUT is an 8-channel paper tape code value, regardless of whether the data tape is 5-channel or 8-channel.

However, 5-channel tape may only be read through input channel 1, i.e. tape reader 1, or reader 2 + SELECT.

The following conversions apply to 5-channel characters which do not exist in the 8-channel code:

.	-	:
@	-	$_{10}$
lf	-	L
cr	-	ignored, a new character is read
fs	-	is interpreted, a new character
ls		is read.

K. KÄRKKÄINEN,
FINNISH STATE
COMPUTER CENTRE

February, 1966.

503 AUTOCODE OPERATION SHEET

DESCRIPTION

TITLE..OPERATOR ....

PROGRAMMER...Tel. No. ....

DATE SUBMITTED..DATE RUN................

EXPECTED TIME................MAX. TIME................ACTUAL TIME............

TRANSLATION

1. Entry Point .... 2. Trace required YES/NO

3. Checks required YES/NO 4. No. of program tapes...................

RUNNING

1. Trace required YES/NO 2. Checks required YES/NO

3. Entry at reference number................ 4. Program contains WAIT YES/NO

5. Program contains DTWAIT YES/NO

6. Number of data tapes (a) reader 1.................... (b) reader 2........................

7. Continue after errors
   (a) Floating-point overflow YES/NO (b) others YES/NO

8. Selected Stop YES/NO

AT	ACTION

SPECIAL INSTRUCTIONS

OPERATOR'S COMMENTS