CONTENTS LIST.

CONTENTS LIST (continued)

This manual describes the features of the existing Symbolic Assembly Program. Chapters describing further features and the application to a non-basic 503 will be added at a later date.

CHAPTER 1.

This chapter introduces the Symbolic Assembly System, which is the normal medium for 503 Machine Code Programming. Some knowledge of the computer, its machine code and the working of its 'Reserved Area Program' is assumed, and the reader is referred to other parts of the 503 Library for a detailed discussion of these.

The 503 can interpret each of the 39-bit information words in its store as a pair of instructions or as an item of data; each instruction is composed of a function and an address portion. A program containing jump or data handling orders has to refer to the locations it will occupy when in the computer; if it does so by means of the addressing system 'built-in' to the computer ('absolute' addresses), the result will be a rigidly-knit program which must always be read into the same part of the store, and which can be modified only at the cost of numerous small alterations made to every part of it.

References to the locations which are to contain a program written in Symbolic Assembly Code are by means of names invented by the programmer. Used in this way, a name merely reserves a computer location, but no particular location is specified at the time of writing the program. A jump instruction might then be written as '43 ERROR 5' and the particular instruction in the program referred to here as 'ERROR 5' would be specified by prefixing it with that label. Again, the program might refer to the (variable or constant) data stored in a location named 'D1', or 'column'.

Thus:

```
          30 d
repeat)   05 subtracting constant
          22 Count 1
          41 Print
          40 repeat
```

The advantage of such a system is that it takes the routine part of programming out of the programmer's hands, and gives an easily alterable program. For example, it is easier to insert the instruction '26 WORKSPACE' after the jump instruction in

```
          47 a
a)        30 mack
```

than to insert the instruction '26 32' in

47 65: 30 13

(In the first example the jump instruction may be left unchanged, but the jump instruction in the second example must be altered to accommodate the insertion).

A program written in this manner must occupy and refer to particular locations when it is in the store, of course, but decisions about positioning are left to the input program. This routine, the Symbolic Assembly Program, incorporates a system which enables it to marry up names and the addresses of particular storage locations, so that it can translate such a program, and plant it in a part of the store known to be free at the time.

Entry to programs stored in the 503 is normally by way of the 'Reserved Area Program' which reads its control instructions from the typewriter. Once the Symbolic Assembly Program is in the store, a Symbolic Assembly Code tape may be assembled by means of a simple typed instruction. The assembled program will have been given a name, and is run by setting up its data tapes and typing GOTO; ‹PROGRAMNAME›.

Programming errors may be found at runtime by using the Symbolic Assembly Test Program. This is able to trace the course run by an assembled program, print out the contents of specified name-locations at specified points, and compare the stored version of a program with its tape. Errors detected by the last process will only be those caused by the program overwriting itself, since the Reserved Area Program automatically sumchecks any programs entered through its control.

The Symbolic Assembly Code makes use of a number of control names, to clarify program structure (both to the Assembly Program, and as a help to the programmer) and to admit extra facilities. Some are used to class the names invented by a programmer for his program (data, program, block). Others mark the beginning and

end of the program (program, trigger) and the internal division of the program into 'blocks' (begin, end).

The 3 chapters of description which follow describe the essentials of the Symbolic Assembly Code, that is, the amount which a programmer would need to know in order to be able to write an efficient machine code program. Chapter 4 concludes with a programmed example.

CHAPTER 2.

1) Identifiers, their form.

The invented names acceptable within the Symbolic Assembly Code may be composed, in part, of digits. For this reason the term "name", used in the introductory chapter, will be dropped in favour of "identifier". Any combination of letters, digits, and spaces which begins with a letter will be called an "identifier".

For example,

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

Identifiers are distinguished from each other by their first 6 non-space characters. Thus no distinction would be made between "Block111", "Block 1", and "Block 11".

2) Identifiers, when invented.

A Symbolic Assembly Code program normally refers to the storage locations which are to contain it by means of a system of identifiers invented by the programmer.

Thus, each item of variable or constant data used in a program is normally distinguished by an identifier chosen or invented by the programmer. Similarly, those instructions which appear as the destinations of the program's jump instructions are also identified, (with labels). As explained below, in Chapter 3 Section 1, relative addresses may also be used in a Symbolic Assembly Code program.

Reference to the storage locations by means of their absolute addresses is not possible, since the Assembly Program allocates space at assembly time according to the free store then available.

The Symbolic Assembly Code (S.A.C.) recognises

2) Identifiers, when invented. (continued)

that programs are usually written in easily handled sections, or "blocks". Each block must be given a distinguishing identifier, so that cross-references may be made within the program; and the program must itself be identified, so that it can be located at run-time.

These identifiers are used by the Symbolic Assembly Program (S.A.P.) to identify particular locations at assembly time. They do not identify location contents.

3) Identifier Introductions, defined.

As seen in the previous section, the identifiers invented for an S.A.C. program fall into 4 groups:

program, block, data and label identifiers.

To indicate, within an S.A.C. program, to which of these groups an identifier invented for the program belongs is called "introducing" the identifier.

Although all invented identifiers must be introduced somewhere in the program, it often happens that an identifier is used before it has been introduced. The only restriction made is that all identifiers used within a block must have been introduced by the end of the block.

4) Control Names

The control names which have defined meanings within the S.A.C. are composed of lower case alphabetic characters, each preceded by underline. The 6 names here dealt with are:

_program, _block, _data,
_begin, _end, _trigger.

5) Style and Position of Program, Block and Data Introductions.

The introduction of the program's identifier is the

5) Style and Position of Program, Block and Data Introductions.(continued)

first statement made in any S.A.C. program, and the block-identifiers must be introduced immediately after this. Data-identifiers may be introduced anywhere: these are treated below in more detail.

Program, block and data identifiers are introduced by listing them after one or other of the control names program, block, data. Each element in such a list is separated by a comma from its neighbours, separating new lines (denoted by the symbol "L") are ignored by the Symbolic Assembly Program, and the list is terminated by semicolon. (Of course, since program introduces the program's name, there can only be one element in its list!).

The S.A.P. reserves a new storage location for each data identifier it reads. Locations reserved in this way are not "re-used" - that is, they are not associated with new identifiers at any later point in the program.

Arrays are declared as data, their scope being indicated within round brackets after the selected data identifier; for example, "domino (12)". This will cause 13 consecutive locations to be allocated. These may be referred to in the program as

domino, domino + 1, ..., domino + 12,

and "domino" will have been allocated the smallest address in the array.

As an example of identifier introductions, the beginning of a program might be

program	multiple regression;
block	read, print, square root, input, output, control;
data	run, dep vars, indep vars (2), workspace (12).

6) Style of Label Introductions.

Correction of Jump Instructions.

A label identifier is introduced by prefixing it to the instruction which it is intended to identify; it must be followed by a closed round bracket. Each instruction may be given up to 3 labels.

for example

L_1) reallocation) 30 a

These labels may be used in the address part of any instruction within the blocks they are introduced in, and are understood to refer only to the instruction, and not to the instruction pair, which follows them. As the programmer is unlikely to know which location-half any labelled instruction will occupy when in the computer, the S.A.P. has been designed to supervise, and if necessary to correct, those jump instructions which have label identifiers in their address portions.

Thus, if the instruction labelled "reallocation) ..." lay in the 2nd half of a location, and the S.A.P. were to input the instruction "43 reallocation", it would effectively read the order "47 reallocation".

N.B. No provision is made for not changing instructions which have the form of jumps to labelled points, but which are intended for use as parameters.

Labelling may be used as 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 entry

L1) + 0
L2) + 0

in a program causes two locations

6) Style of Label Introductions
Correction of Jump Instructions. (continued)

to be reserved within the program area, and these may be referred to as "L1" and "L2".

7) Label Introductions, continued.

Labels introduced in the way outlined in the previous section (6) have no meaning outside the blocks in which they are introduced. A label has to be given an additional introduction if it is required for use outside its own block. Such labels are called 'global'. They 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.

8) Local and Global Identifiers
Block Identifiers as Labels.

An identifier may be used in more than one program-block, and yet be assigned a different meaning within each of those blocks. For example, the label

ENTRY)

could be fixed to one of the instructions in each block of an S.A.C. program, since a label, unless global, only has a meaning inside the block in which it is introduced.

8) Local and Global Identifiers
Block Identifiers as Labels, continued)

An identifier which has no meaning outside the block in which it is introduced is called LOCAL. An identifier which has the same meaning or reference throughout a program is called GLOBAL. Global identifiers are distinguished from local identifiers by being introduced outside (i.e. between or before) the program-blocks.

Blocknames are naturally global, whereas data and label identifiers may be global or local, or both. Thus, one or more of the "ENTRY" labels referred to at the beginning of this section could be introduced globally at the head of the program. (In blocks within which an identifier has both global and local references, the local reference is understood to hold).

Block-identifiers may be used as if they were labels attached to the first instructions of their blocks, (but see the restriction mentioned in Chapter 3 Section 4). Thus, the instruction "40 blockname" is a jump to the first instruction of block "blockname". Again, the identifier "A" could be used globally as a blockname and also as either a local data or a local label identifier (but not both within any one block) within any of the program's blocks.

9) Form of reference to Global Labels
Relative Identifiers.

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

Local data and label elements, (and also global label elements), are referred to by means of their own identifiers, inside their own blocks. Outside their blocks, global label elements are referred to by means of relative identifiers.

9) Form of reference to Global Labels Relative Identifiers, continued.

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. (See Section 7).

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;

for example, causes 10 distinct locations of data space to be reserved at assembly time. Outside blocks B1 and B2 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 do not have to 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);

10) Block Format.

Each of the instructions and words in an S.A.C. program must lie on one, and only one, block. This means that blocks may not intersect or be contained in each other, and that every program must contain at least one block. The limits of a block are defined by the control names, begin and end, begin being followed by the block's identifier and a semicolon, and end being followed by a semicolon. (All characters between end and the next semicolon will be ignored by the Assembler).

Identifiers may be introduced between any two instructions or words in a block, but it is convenient to gather all data introductions into the "head" of the block, after begin. All identifiers referred to in a block must have been introduced by the time the end of the block is reached; it is enough if global labels have been introduced globally at this time. Labels may be referred to before they are introduced, of course. It should be noted that the contents of a block's data locations are not overwritten or destroyed on exit from the block, since data space is not duplicated.

Thus a block has the form,

```
begin   blockname;
data    a,b,c,d, ...;
(label and data introductions, instructions
 & words)
end;
```

No block may occupy more than 4096 locations when in the store.

11) S.A.C. Program Format.

3 parts may conveniently be recognised in each S.A.C. program.

(a) The program begins with a "head", containing the introduction of the program's identifier, followed by the introductions of the block and (global) data identifiers. An example of a program head is given at the end of Section (5).

11) S.A.C. Program Format, continued

(A minimum program head contains the introductions of the program's identifier, all block identifiers, all global label identifiers, and the global data identifiers used within the first block).

(b) The blocks, with global data introductions between them, if necessary.

(c) The tail. 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:

trigger lift*block 1, stairs*block 1, flats;

The S.A.P. causes each S.A.C. program it assembles to be followed by a trigger block of entry points. This contains 3 + n words, where "n" is the number of entry points.

An example of a complete S.A.C. program is given at the end of Chapter 4.

5) Restrictions on addresses

Unallocated identifiers and Blocknames used as labels may not be used in COMPOUND Addresses, and 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.

CHAPTER 3.

This chapter deals with the orders and wholewords which go to make up an S.A.C. program.

1) Instructions.

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

The addresses may be of 5 different types.

(a) INTEGERS without signs, e.g. "51 5". 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 15th location in this block".

(c) IDENTIFIED, e.g. "30 shovel"

(d) COMPOUND

(e) DIAMOND-BRACKETED

Addresses of types (d) and (e) are dealt with in Section 3.

2) Wholewords.

The content of a location may be expressed in different ways. Any S.A.C. expression of a 39 bit location content is called a "wholeword".

2) Wholewords (continued)

Wholewords are divided into the following types:

Integers, Fractions and Floating Point Numbers,

Octal & Alphanumeric Groups,

Order Pairs,

Signed Identifiers.

2.1) Integers, Fractions and Floating Point Numbers.

These are composed of the digits 0 to 9 and the characters + - . and subscript$_{10}$. They may be terminated by any character other than these, and fractions and integers may also be terminated by + and -
Each number must contain at least one digit and be preceded by a sign.

Any whole number between -274,877,906,944 and +274,877,906,943, inclusive is called an INTEGER

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 the S.A.P. 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.

2.1) Integers, Fractions and Floating Point Numbers - (continued)

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:

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

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.

Example: Zero may be expressed as

+0 -0.0 $_{10}0$ .0 $+0_{10}$ +00000 and so on.

But ".0000000000000" is erroneous, because it is a fraction with more than 11 digits, and so is "+" because it holds no digit.

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".

2.2) Octal Groups (continued)

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

2.3) Alphanumeric Groups

It is often useful for an S.A.C. program to be able to output headings, titles, error indications, etc. on the typewriter at runtime. Such headings are to be stored in some way, as constant data.

The 7-hole character code is punched in an 8-hole form, on tape, one of the bits in each character being used for parity checks. Five such characters 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:

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

The S.A.C. expression for the above location content 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 "paperthrow" (P), which

2.3) Alphanumeric Groups (continued)

have no effect on print-ups.

Thus the expression for

for heating: £96

as a set of alphanumeric groups is

£forSh£catin£g:S£9£6PPPP

where S denotes the character "space", and P 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 S.A.C. program which follows will output the word ERROR on the typewriter.

```
        data count, press;

          30  error A
          50  35
          30  Nm4
          20  Count
again)    06  0
          54  7
          20  Press /
          74  4096
          32  Count
          41  Again

          ..............

errorA)   £ERROR
  Nm4)    -4
```

2.4) Instruction Pairs.

An instruction pair is a pair of instructions separated by a B-line.

2.4) Instruction Pairs (continued)

Example. The number 100 may be expressed as

+100
80 000 000 000 124
000:00 100

in the S.A.C.

2.5) Signed Identifier Wholewords

The effect of writing

+ IDENTIFIER

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

3) DIAMOND BRACKET Addresses
COMPOUND Addresses.

Whole words may be stored in an S.A.C. program by actually writing them as part of the program, and normally with a label for identification.

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

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

An alternative way of doing this is by the order

$30\langle+3.14159265_{10}\rangle$

The S.A.P. 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 they are first read in; they are not duplicated in the store if read again, and are available throughout the

3) DIAMOND BRACKET Addresses
COMPOUND Addresses - (continued)

program.

Example: the instruction

30 ⟨+ count⟩

has the effect of placing the address of the location allocated to "count" in the accumulator.

A COMPOUND address consists of an identifier address with an added integer.

Example 30 pick +3

or 30 3+ pick

mean "pick up c (pick +3)" and do not mean "Pick up c(pick) +3".

4) Allocation.

The process by which the S.A.P. determines the reference of an identifier is called "allocation".

For example, the data location "DATA 15" will have been allocated when the S.A.P. 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. The one exception is that, when an S.A.C. program is being output, local data introduced outside the head of a block is allocated when the S.A.P. reaches the end of the block. 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.

CHAPTER 4.

This chapter concludes the description of that part of the S.A.C. needed to write a complete S.A.C. program. It deals chiefly with those identifiers, not control-names, which the S.A.P. recognises as having a fixed meaning. The chapter ends with an example of a complete S.A.C. program.

1) Subroutines.

The S.A.P. 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 S.A.P. 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 : the S.A.P. 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

1) Subroutines (continued)

entry to the subroutine, may be shortened from

```
      00  LINK / 40 n
to        EXIT, n
```

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.

2) Common Programs.

Each S.A.C. program is automatically allocated a link location by the S.A.P., so that S.A.C. 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 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 common program itself.

2) Common Programs (continued)

It is important that program X be already in the store when S.A.P. is assembling the program which calls it.

3) Style of Writing S.A.C. 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 the S.A.P., but, from the programmer's point of view, they are more difficult to alter than those in the suggested style.

The S.A.P. 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 /
```

would be stored in consecutive locations.

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

3) Style of Writing S.A.C. Programs
B-digits, dynamic stops, comments (continued)

These are written as

STOP - unconditional transfer to the R.A.P.

O STOP - transfer if overflow to the R.A.P.

N STOP - transfer if negative accumulator to the R.A.P.

Z STOP - transfer if zero accumulator to the R.A.P.

The S.A.P. will ignore comments punched between square brackets, so that "30 A [A = collating const.]" is read as "30 A".

4) Example of an S.A.C. Program.

```
program  square root;

[this program computes the roots of single length
fractions and precedes the output result with a
new line and the word ROOT]

blocks 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
        20 number*root
        30  £LROOT
        COMP, alphanumeric output
```

4) Example of an S.A.C. Program (continued)

```
        SUBR, root
        COMP, print, 2
        000 / 0011  [Print parameter word]
        40    control
   L1)  30   (£NEGPP)
        COMP, alphan
        40    control
    end;
    trigger  control;
```

27th February, 1963.

P. Schwar.