Volume 2: PROGRAMMING INFORMATION

Part 1: PROGRAMMING LANGUAGES

Section 1: MASIR

Contents

Page

Chapter 1: INTRODUCTION

1.1 Purpose 1
1.2 Features of MASIR 1
1.3 Configuration 2

Chapter 2: MASIR LANGUAGE AND COMPATIBILITY WITH SIR

2.1 MASIR Relationship to SIR Facilities 3
2.2 Definition of MACRO 3
2.3 MASIR Language - Text Processing Facilities 3
2.4 MASIR Language - Code Generation Facilities 4
2.5 Structure of a MASIR Program System 4
2.5.1 MASIR Program Unit 5
2.5.2 Blocks 5
2.5.3 Identifiers and Labels 6
2.5.4 Global Identifier Lists 7
2.5.5 Local Identifiers 8
2.5.6 Example of Block Structure 8
2.6 Words 9
2.7 Instructions 9
2.7.1 Absolute Addresses 10
2.7.2 Relative Addresses 10
2.7.3 Identified Addresses 11
2.7.4 Literal Addresses 11
2.8 Quasi-Instructions 12

Page

			Page
	2.9	Constants	13
		2.9.1 Integers and Fractions	13
		2.9.2 Octal Groups	13
		2.9.3 Alphanumeric Groups	14
		2.9.4 Pseudo-Instructions	14
	2.10	Skips	17
		2.10.1 Labelled Skips	17
		2.10.2 Repeated Data	18
	2.11	Comments	18
		2.11.1 Titles	18a
	2.12	Patches	18a
		2.12.1 Absolute Patches	18a
		2.12.2 Module Patches	18b
Chapter 3:	MACRO FACILITIES		
	3.1	General Description	19
	3.2	Definition of MACROS	19
		3.2.1 CALLG(X)	20
		3.2.2 MQCHOP	20
		3.2.3 QFPCALL	21
	3.3	MACRO Calls	21
	3.4	Numeric Label Generation by *$	22
	3.5	Addition to MACRO Text	23
Chapter 4:	ASSEMBLY FACILITIES		
	4.1	Conditional Assembly of Text	25
	4.2	The Use of FUNCTION MNEMONICS	26
	4.3	Additional Addressing Facilities in MASIR	27
	4.4	MASIR Subroutines, QPARAM	30
Chapter 5:	SUMMARY OF MASIR DIRECTIVES		
	5.1	*DEFINE	33
	5.2	*DELETL	33
	5.3	*GDEFIN	33
	5.4	*ADDMAC	33
	5.5	*IF	34
	5.6	*IFNOT	34
	5.7	*LISTLA	34
		5.7.1 *NOLIST	34
	5.8	*CHECKW	34
	5.9	*PROG	35
	5.10	*NOMEM	35
		5.10.1 *MEM	35
	5.11	*CHANGE	35
	5.12	*SETDIC	36
	5.13	*NOTITLE	37
		5.13.1 *TITLE	37

Page

Chapter 6: MASIR OPERATING INSTRUCTIONS

6.1 Form of Distribution 39
6.2 Assembler Operating Instructions 39
6.3 List of Assembler Options 40
6.4 Loader Operating Instructions 41
6.4.1 Function 41
6.4.2 Distribution 41
6.4.3 Operating Procedure 41
6.5 Entry to Program 42
6.6 List of Loader Options 42
6.7 Loader Setting for Various Store Sizes 43

Chapter 7: STORE USED

7.1 MASIR Store Used 47
7.2 Store Used by Loader 47
7.3 Store Used by Loader Programs 47

Chapter 8: ERROR INDICATIONS

8.1 MASIR Error Reports 49
8.2 Loader Error Reports 52
8.3 905 MASIR Example Program 54

Appendix A: SYMBOLIC INPUT ROUTINE (SIR) App. A 1 - 52 incl.

Appendix B: DIFFERENCES BETWEEN MASIR AND SIR FACILITIES App. B1

Appendix C: MAPLOD (LABEL LISTING PROGRAM) App. C 1 - 2

905 MACRO ASSEMBLER (MASIR)

Chapter 1: INTRODUCTION

1.1 Purpose

The Macro Assembler Program for the SIR Language (MASIR) provides the power associated with a machine code language whilst retaining many of the programming advantages normally associated with a higher level language. The assembler, which allows a large number of user-defined MACROS in addition to conditionals and source lines, generates relocatable binary code which may be loaded into any store module by a linking loader.

The SIR Assembler (a subset of MASIR) is described in Appendix A to this section of the manual.

1.2 Features of MASIR

The Programming features of MASIR include:

(a) USER DEFINITION OF MACROS with replaceable parameters and nested definitions (if necessary).

(b) CONDITIONALLY COMPILED CODE and MACROS which allow dummy peripheral routines or diagnostic information to be assembled, or ignored, without the editing of a source program.

(c) MACRO CALLS WITH PARAMETERS enable a large number of machine code instructions to be generated by a single macro instruction.

(d) MNEMONIC NAMES for MACHINE CODE FUNCTIONS as alternatives to numeric function codes used by the SIR Assembler (Appendix A).

(e) FLEXIBLE LOADING OF PROGRAM UNITS into any store modules and communication between these units. Communication is made possible by global labels and special Assembler/Loader macro features.

(f) ABILITY TO COMBINE FORTRAN and ASSEMBLY CODE enables programs to be linked during loading by the linking loader.

(g) COMPATIBILITY WITH EXISTING 900 SERIES SIR PROGRAMS. The existing facilities of SIR are included in MASIR with the exception of those items that are only usable in load-and-go mode, and options which are replaced by directives (See Chapter 5).

1.3 Configuration

The assembler will run on any current 900 Series machine, i.e. 903, 905, 920B, 920C, 920M. The basic version requires 8K store (8192 words), paper tape reader, paper tape punch and teleprinter.

Chapter 2: MASIR LANGUAGE AND COMPATIBILITY WITH SIR

2.1 MASIR Relationship with SIR

MASIR is an Assembler program containing macro facilities for the 900 Series 18-bit computers.

MASIR has, compared with SIR many additional facilities (see Chapters 3, 4, 5).

The SIR assembly language is described in Appendix A and the differences between MASIR and SIR facilities are listed in Appendix B to this manual.

2.2 Definition of MACRO

A MACRO is a string of characters (which are made up of any sequence of allowable characters including English text, machine instructions, data items, macro definitions and calls, etc.) associated with a given name (MACRO NAME) which is inserted in the text of a program wherever that particular name is used in a macro call. This string may be modified by replacing parts of it by actual parameter strings specified in the call.

Examples:

(1)

```
*DEFINE READCH (X)
[
 4        +0
15        2048
 5        X]
```

(2)

```
*DEFINE DATA (X)
[X,X₁₀3, X₁₀-2]
```

(3)

```
*DEFINE MAC2 (D)
[
* DEFINE  READCH (Z)
[
15        D
5         Z
]
]
```

2.3 MASIR Language, Text Processing Facilities

MASIR has two processes, viz:

(i) text processing facility

(ii) one-to-one assembling.

The MASIR language consists of a string of characters forming 'text' which may contain some, all or none of the following:-

MACRO definition (Chapter 3.2)

MACRO calls (Chapter 3.3)

Facilities associated with MACROS (Chapters 3.4 and 3.5)

Conditionals (Chapter 4.1)

The MASIR assembler passes this 'text' through a MACRO generation stage, which produces on output a string of characters 'text' but which exclude all the text processing facilities. Under normal assembly conditions this 'text' is not actually output to an external receiver, but in fact is passed direct to the code generator stage of the assembler program. By using a suitable option (see Chapter 6) this intermediate text may be output, a check can then be made to see whether the processing of macro text is as expected.

The macro generator facility may be used independently as a text processor, to produce any suitable text output; for example, macro generator of repetitive data, macro generation of Fortran programs etc.

2.4 MASIR Language - Code Generation Facility

The 'text' produced using the text processing facility, is passed to the code generator which should form a valid MASIR program or programs. In this 'text' there is normally a one-to-one relationship between elements of text and words of code generated. The code generator converts this 'text' into RLB program words, which is then loaded into store as a fixed program with fixed operand addresses.

2.5 Structure of MASIR Program System

The remainder of this chapter describes MASIR text and the associated rules used after processing the MACRO generation (text processing) facility. For example, if it is stated that only identifiers, separators and " symbol can be included in a global label list, it then follows that the original may contain MACRO calls and definitions within the global list.

2.5.1 MASIR Program Unit

A MASIR program system consists of one or more program units (the name "program unit" is frequently abbreviated, to "program"). A program unit may be headed by the directive *PROG, which is followed by a name (allocated to the first location of the program unit), and must be terminated by a newline % newline.

Each program unit is assembled independently of other units, and the code generated from this unit must be loaded into one store module. Therefore code generated plus local workspace must not exceed 8192 words.

A program unit should be made up from the following elements after text processing:

- Directives
- Global label lists
- Blocks
- Labels
- Words
- Skips
- Comments
- Patches
- Percent Line (equivalent to the sequence newline, percent, newline)

In general MASIR words bear a one-to-one relationship with the core store words occupied by the program when it is loaded (see Appendix A Chapter 1.3).

2.5.2 Blocks

A MASIR program unit consists of one or more blocks. Each block begins with a global label list, and is terminated by either the start of a new global label list, or a percent line. Between the global label list and the terminator a block may be made up of the following elements:

- Directives
- Patches
- Skips
- Comments
- Labels
- Words

The division of a program unit into blocks is under the control of the programmer. Block structure may be used:-

(a) To limit the scope of label declarations, so that identifiers may be inserted freely without fear of inconsistency.

(b) To divide a program into convenient sections so that later it can be easily interpreted by others.

(c) To clarify which parts of the program communicate with other program units.

2.5.3 Identifiers and Labels

An identifier is a name invented by the programmer. It may be the name of a macro, a program unit, a data item, an array or program instruction.

Identifiers can be devised using any combination of the alphabetic characters and the digital characters 0-9, but the first character of any identifier must be an alphabetic character. Identifiers are distinguished from each other by their first six characters only. (See Appendix A Chapter 2.1). Identifiers may be declared in a MASIR program unit in one of five ways:

(i) Use in a *DEFINE, *G DEFIN, *ADDMAC, for macro names (See Chapter 3 and 5).

(ii) Use in a *PROG directive. This effectively declares the name as a global label labelling the first location of the unit, see Chapter 5.9.

(iii) Use in a global label list.

(iv) Use as labels to instructions or data items (constants, quasi-instructions or skips).

(v) Use as labels in the form LABEL = absolute address.

Note that SIR and MASIR do not make any distinction between identifiers (labelling instructions) and those used as data identifiers. They are declared in the same way, by labelling a word.

An identifier used as a label is written on its own,preceded and followed by space or newline separators. It then becomes associated with the address, into which the word following that label would be assembled.

A label in the form Label = absolute address indicates,that label is associated with that address. Any absolute addressing be used and it may be written as m or m ↑ n where m<8192 and

n is a module number ($n < 16$)

e.g. LABEL 2 = 200 ↑ 1

This means that LABEL 2 labels the location 8392 (200 + 8192) and does not imply setting a value into LABEL 2 location. For examples and further details see Appendix A Chapter 2.1.

2.5.4 Global Identifier Lists

The start of a block is signified by a Global Identifier List enclosed in brackets [ ]. This list may only contain identifiers, separators or double quotation marks. Global Identifiers and their uses are described below.

Global Identifiers form the link between the program units. Sub global identifiers form the links between different blocks of a program unit. They must be listed in the Global Identifiers Lists at the head of:

(a) the block in which they are declared

(b) every other block in which they are to be valid. Each global or sub-global must occur as a label once in the total area in which it is valid.

One or more separators must follow each identifier in a Global Identifier List; only identifiers, separators and Sub-Global Identifier markers (") may occur between the brackets which enclose the list. When an identifier is included in the Global Identifier Lists of two or more blocks which are assembled together, that identifier refers to a single address (indicated by a label in one of the blocks - namely, the block in which it is declared). An identifier used 'globally' in some blocks may be used 'locally' in any block in which it is not listed as global.

The name of a program unit is automatically global to other units. It can also be declared as global in its own unit and can then be referenced from within the unit.

Sub-Global Identifiers are signified by the use of the double quotes " symbol. If on its first occurrence in a Global Identifier List an identifier is preceded by the " symbol, it is treated as sub-global. Whereas, a Global Identifier is passed on to the relocatable binary loader (thus permitting communication between several program units, held jointly in store), Sub-Global Identifiers are removed from the MASIR dictionary when % is encountered. The listing of an identifier as Global or Sub-Global is determined by the first Global Identifier List in which it occurs and is valid for a complete program. An identifier cannot be Global in some blocks of a program and Sub-Global in other blocks of that program. Once an identifier is Sub-Global, the use of ", before further references in global lists is optional.

Examples of Global and Sub-Global Identifiers.

MOUSE"HAMSTER"LION WOLF

MOUSE AND WOLF are Global Identifiers

HAMSTER AND LION are Sub-Global Identifiers.

2.5.5 Local Identifiers

Identifiers which are neither Global or Sub-Global are termed Local and have no meaning outside the block in which they are declared.

A name (or identifier) may be used to represent a Global or Sub-Global Identifier in some blocks, several different Local Identifiers in other blocks, and be undefined elsewhere in a program.

Each Local Identifier is declared by being used once and only once as a label in the block for which it is valid. Similarly each Global or Sub-Global Identifier is declared by being used once only as a label in only one of the blocks for which it is to be valid.

2.5.6 Example of Block Structure

```
*PROG    PREDICT
[ PREDICT SPEED DIST "B2]
                +0
        ST      W
        MUL     SPEED
        SH      17
        ST      DIST
        J       B2
W               +0
[ B2 PREDICT]
B2      ST      DIST
        LDB     PREDICT
        J/      1
DIST            >1
%
*PROG  CALC
[ CALC PREDICT "B2 SPEED DIST]

        CALLG (PREDICT)
        8       B2
SPEED           +0
DIST            +0
["B2]
B2      8       ;+0
%
```

The example is not realistic.

PREDICT is Global to program CALC
DIST is Global in block PREDICT
Another DIST is local in block B2 of PREDICT
B2 is Sub-Global in program PREDICT
Another B2 is Sub-Global in program CALC

2.6 Words

Words are the basic elements of a MASIR program. They can be written in several forms: constants, instructions, or special address forms (see Chapter 4.3).

For example:

+ 304 and
- .2667

are constants, whereas:

15 2048 and
/2 CAT+10

are instructions.

+ LOCN

is an address form.

Each word must be followed by a separator character. On assembly a SIR word will occupy one store location within the core store. Words are entered into consecutive store locations in the order that they appear in a program unless the Assembler receives a directive (e.g. patch, skip, etc.) to the contrary.

2.7 Instructions

Words written in the form of instructions are introduced by a /(solidus) character or a digit. Each word consists of a function part and an address part, which are separated by one or more separator characters (e.g. space).

If the solidus precedes the function part this indicates that the address part is to be modified by the contents of the B register. The function part consists of a decimal integer in the range 0 to 15; each integer represents a 900 machine function (e.g. 4 represents the function load the accumulator). Alternatively the function part may be a mnemonic, see Chapter 4.2. The address part can be written as Absolute, Relative, Literal, or Identified (defined in 2.7.1 to 2.7.4). An address is assembled as an integer in the range 0 to 8191.

If a function mnemonic is used, the solidus representing modification may be placed before or after the mnemonic.

2.7.1 Absolute Addresses

An absolute address consists of an unsigned integer not greater than $(8191)_{10}$ and refers to the core store location with that integer as its address. In machine code functions 14 and 15, the absolute address provides further specification of the function using standard conventions.

Examples of absolute addresses are:

4 8180	Meaning, load the accumulator with the contents of location 8180.
15 6144	Meaning, punch the least significant 8 bits of the accumulator.

2.7.2 Relative Addresses

A relative address can be one of two types, a 'location' relative address or a 'block' relative address. Integers used in relative addresses must be in the range 0 to $(8191)_{10}$.

A location relative address consists of a semicolon,followed by a signed integer,and refers to a location the address of which is:

The address in which the current instruction is being assembled + the signed integer value.

Examples of location relative addresses are:

7 ;+3	Meaning, jump three locations forward if zero.
5 ;-1	Meaning, store in the previous location
8 ;+0	Meaning,perform a dynamic stop.

NOTE: 8 ;0 is an invalid instruction as the integer following the semicolon is unsigned.

A block relative address consists of an unsigned integer not greater than 8191 followed by a semicolon and refers to a location with an address equal to:

The value of the unsigned integer + the address of the first location in the current block.

Examples of the use of block relative addresses follow.

[ONE TWO]	Two global addresses
START +1) +2)	Constants
4 0;	Load the accumulator with the contents of location (0 + START) = +1
5 ONE	Store contents of accumulator in ONE
4 1;	Load the accumulator with the contents of location (1 + START) = +2
5 TWO	Store contents of accumulator in TWO

2.7.3 Identified Addresses

An identified address consists of either, an identifier, or an identifier followed by a signed integer. An identified address is introduced by a letter.

The assembler will replace the identified address with the sum of the absolute address of the location (labelled in a unique manner by the identifier) and the signed integer (called an increment - even if negative in value). The increment must be in the range +4095 to -4096, and the address formed by identifier + or - increment must lie in the same store module as the Identifier.

An identified address can be used in the TEXT prior to the declaration of the identifier to which it refers (i.e. prior to the identifier appearing as a label).

2.7.4 Literal Addresses

Literal addresses are introduced by any one of the following symbols:

+, -, =, &, or £.

They are used to make it easier to write instructions which operate on constants. Their function is indicated in the examples which follows:-

Example 1

```
TEN     +10
        ........
        4 TEN
```

In this instance the identifier which labels a constant to be used at some other point in the program is placed in the address part of the instruction. Whereas:

Example 2

4 +10

in this example the programmer simply places the constant itself into the address part of the instruction. During assembly, the assembler on reading the end of program symbol % allocates a store location to the constant, places the constant therein, and finally, inserts the address of this location in all the instructions using this constant.

There are four types of literals corresponding to the four types of constants available to MASIR. These literals are:

Integers and fractions

Octal groups

Alphanumeric groups

All of which have the same format as their corresponding constants for example,

4 -.2667	Fraction	6 &7777	Octal
2 +360	Integer	4 £E5 ↑	Alphanumeric

and finally the literal type:

Quasi-instructions (detailed in the next section 2.8)

Module address, relative and module address absolute may also be used in literals (see Chapter 4.3).

2.8 Quasi-instructions

These literals are similar to their corresponding pseudo-instruction constants (Section 2.5.4) but differ from them in the following manner:

(1) Every quasi-instruction is introduced by the symbol = which immediately precedes the function bits or solidus (indicating B register modification when present).

(2) The address part of a quasi-instruction must be in absolute form (relative, identified or literal addresses are signified as errors by the error message ERROR IN LITERAL).

Examples of Quasi-instruction literals are:

4 =8 0 Load accumulator with the constant $2^{16} = 65536$

6 =15 8191 Collate the accumulator with 131071 (binary 1 less than 2^{17})

2.9 Constants

Five types of constants are available to MASIR. They are:

Integers and Fractions

Octal Groups

Alphanumeric Groups

Pseudo-instructions

All constants must be followed by a separator character.

2.9.1 Integers and Fractions

These are introduced by a + or - sign. If the + or - sign is immediately followed by an integer then the constant is stored as a binary integer. Viz.

+ 14 is stored as 000 000 000 000 001 110

- 64 is stored as 111 111 111 111 000 000

Integers must be in the range -131071 to +131071 inclusive. The integer - 131072 may be written as the pseudo-instruction /0 0 or as the octal group & 400000.

If a + or - sign is immediately followed by a decimal point (·) then one or more digits, the constant is stored as a binary fraction. For example:

+ ·375 is stored as 001 100 000 000 000 000

and - ·5 is stored as 110 000 000 000 000 000

The fraction - 1 can be written in the same manner as the integer -131072.. Fractions can contain up to six digits.

2.9.2 Octal Groups

These groups are introduced by the symbol &. An 18-bit 903 word can be divided into 6, 3-bit groups each group being equivalent to a digit from 0 to 7. Thus a constant can be written as an & symbol followed by a group of 6 octal digits. For example:

&312705 can be written to represent the binary pattern:

011	001	010	111	000	101
3	1	2	7	0	5

Octal groups of less than 6 digits can be written and in this event the digits are right-justified, i.e.

$$\&42 \equiv \&000042$$

2.9.3 Alphanumeric Groups

These groups are introduced by a £ symbol which immediately precedes three alphanumeric characters. These characters are packed (from left to right) into a store location in 6-bit SIR internal code. This internal code is as follows:

900 6-bit Internal Code

External Character	6-bit Code (Octal)	External Character	6-bit Code (Octal)
Space	00	0	20
Newline	01	1	21
"	02	2	22
½ or £	03	3	23
$	04	4	24
%	05	5	25
&	06	6	26
´ acute	07	7	27
(	10	8	30
)	11	9	31
*	12	:	32
+	13	;	33
,	14	<	34
-	15	=	35
.	16	>	36
/	17	10 or ?	37
\ grave	40	Pp	60
Aa	41	Qq	61
Bb	42	Rr	62
Cc	43	Ss	63
Dd	44	Tt	64
Ee	45	Uu	65
Ff	46	Vv	66
Gg	47	Ww	67

contd.

External Character	6-bit Code (Octal)	External Character	6-bit Code (Octal)
Hh	50	Xx	70
Ii	51	Yy	71
Jj	52	Zz	72
Kk	53	[	73
Ll	54	£ or \	74
Mm	55	]	75
Nn	56	↑	76
Oo	57	←	77

NOTES: 1. On input no distinction is made between upper and lower case letters. Letters are always output in upper case (i. e. caps).

2. Newline is a compound symbol consisting of the CR and LF characters. The MASIR Input Routine ignores carriage return (CR) but recognizes line feed (LF) as significant.

3. Tab is equivalent to space.

4. The alternatives shown will be used on I. S. O. code teleprinters.

In packing alphanumeric groups all characters in the table given can be stored with the exception of the following:-

(a) % cannot be included (end of program symbol. See 2. 9. 2).

(b) ↑ and ← are stored as the octal number 01 (i. e. code for newline) and not in their own codes (Octal 76 and 77).

(c) An alphanumeric group is considered complete if a newline is encountered before three characters have been read after the £ symbol; in this event the group is left-justified (i. e. the remaining characters are considered to have the code 0, the code for a space character).

Newline is NOT considered as a character within the group but acts as any other normal separator. Spaces which occur in the three characters following the £ symbol are treated as normal characters.

The prime function of alphanumeric groups is for storing characters which are to be punched out during a program run. It must be noted that this operation can only be performed when the program using this function contains a print routine and a table for conversion from internal to external code.

Examples of Alphanumeric Groups

Group Required	Form in Store	
	Octal	Alphanumeric Equivalent
£ MAN	55 41 56	MAN
£↑ H←	01 50 01	Newline H newline
£ space = newline	00 35 00	Space = Space

NOTE: The spaces in octal equivalents are for clarity purposes only. They must NOT be punched.

Since alphanumeric groups containing ↑ and ← cause newline code to be stored, if a representation of ↑ and ← is necessary an octal group must be used. Viz.

Group Required	:	Octal Equivalent (as written)
↑←↑		& 76 77 76
A %←		& 41 05 77

2.9.4 Pseudo-instructions

These are identical in format to ordinary instructions, but are used as constants, for example, /0 0 can be used to represent the integer -131072.

Similarly, it is possible to obey constants as instructions though the intentional use of this effect is NOT recommended. A failure to terminate an instruction sequence with an unconditional jump (for example, a dynamic stop) is liable to result in this undesirable effect.

2.10 Skips

A skip signified thus >, indicates that during assembly a specified number of store locations are to remain unaltered before the MASIR Assembly continues filling the store with words. The number of store locations to remain unaltered is specified by an optional + sign and an integer. The specification characters immediately follow the > symbol.

Consider the following example of a skip:

+ 133	Word
>+15	Skip
4 8180	Word
5 COUNT	Word
....	Further coding

If in this example the word + 133 was entered into location 5000 of the Core Store, the Skip (> + 15) indicates that the next word (the instruction 4 8180) is to be assembled in location 5016 and not 5001. 5 COUNT would then be assembled in location 5017 etc.

The prime function of skips is in reserving locations for work space without assigning any values to those locations.

2.10.1 Labelled Skips

Locations left unchanged by skips may be labelled in the same manner as locations occupied by words. For example:

	8 ERROR
	> +4
ALPHA	>+10
MATRIX	>400
BETA	>10

In this example, if 8 ERROR is assembled in location 4000, ALPHA refers to location 4005, MATRIX to 4015 and BETA to 4415.

NOTES: (a) The last word of the 10-word vector labelled ALPHA is addressed as ALPHA+9. Similar addressing applies to MATRIX and BETA.

(b) Addresses outside the range indicated in (a) can be referred to by incremented instructions. Thus ALPHA +11, MATRIX+1 and BETA -399 are alternative ways of

(b) contd. referring to the second location of the array MATRIX. If the length of ALPHA was changed, the increment relative to ALPHA would have to be changed. Similarly, if the length of MATRIX was changed the increment relative to BETA would also have to be changed.

2.10.2 Repeated Data

This facility can be used to set a particular constant or instruction into a number of consecutive store locations. It is written as follows:

```
>n:data
```

where data is any valid MASIR element. It is assembled as though n items of source code has been given with the character string defined by data.

E.g.

```
>10:+0
>6:-1
>11:&014367
>5:0   LABEL
>12:10 ERR
>8:0 ;+0
```

The form:

```
> m: NAME data
```

is equivalent to:

```
NAME > m: data
```

For example:

```
>4:ARRAY1 +1
```

would be assembled as:

```
ARRAY1  +1
        +1
        +1
        +1
```

2.11 Comments

These are included in a program to make the printout of that program easier to understand.

Any string of characters (valid in the internal code) enclosed in parenthesis ( ) is ignored by the MASIR code generator.

2.11.1 Titles

These may be included in a program to indicate the progress of assembly, or to identify a particular version or revision of a program on the assembly listing. They are optionally displayed on the teletype at assembly time. Titles should be enclosed by double round brackets.

E.g. ((TAPE 2 VERSION 6 18-10-71))

Any characters valid in the internal code may occur between the double brackets.

Title listing can be suppressed by the directive:

*NOTITLE

and allowed again by the directive:

*TITLE

Titles have no other effect on the assembly of the program.

2.12 Patches

A patch is a special form of directive which directs the Loader (via the assembler) to store a program unit in a particular part of store, instead of storing the contents in the next free position.

Two types of patches are available:

1. Absolute patches
2. Module patches

2.12.1 Absolute Patches

These specify the actual location at which the next word of a program unit is to be stored.

2.12.1.1 Absolute address patches

These are written in the form:

↑ address

or

↑ label

or

label ± increment

Any label used must be already located in this unit.

A label may only be used if not code has yet been laid down, or if address patching is already in use.

Address patches can be used as often as desired, throughout a program unit, provided that no module boundaries are crossed.

2.12.1.2 Absolute global patches

These are written in the form:

↑ global label

or

↑ global label ± increment.

This can only be used if no code has yet been laid down and the label is not located in this unit. An error message will be given if the label is subsequently located in this unit.

The label must have had an absolute address allocated to it before the patched unit is loaded.

For all absolute patches, the loader does not update its free store pointers, so the user must exercise some control over the layout.

2.12.2 Module Patches

These specify the module into which a program unit is to be loaded, without fixing its address within that module.

They are written in the form:

↑↑ module number

or

↑↑ global label

If a global label is used, it must have had an absolute address allocated to it before the patched unit is loaded. The program will then be stored in the module which contains the global label.

The program unit will be stored downwards from the next free location in the module specified. If there is not enough room, the loader gives an error. Loader free store pointers are updated.

Chapter 3: MACRO FACILITIES

3.1 General Description

The Macro Assembler contains facilities for writing single statements which can generate several machine code instructions; for example, one statement can generate a two word subroutine call, or a collection of Macro definitions can be grouped so as to resemble a high level language, orientated to a particular application. When using the latter method, the need to revert to machine code notation is eliminated.

3.2 Definition of Macros

Macros are defined by the directive *DEFINE in the form:

*DEFINE (s) MACRONAME (Formal parameter list) [TEXT]

(s) = Space (which may be replaced by a sequence of space or newline characters. Any characters between MACRONAME and Formal parameter list and/or text will be ignored, this includes * $.)

A MACRONAME represents a name chosen by the user, in the form of an identifier.

The 'formal parameter list' consists of identifiers separated by commas. However if there are no parameters to be listed then the parenthesis ( ) must be omitted.

TEXT may consist of any string of characters allowable in SIR internal code (see Appendix A Chapter 2.5.3).

The text must always be enclosed within a square bracket pair i.e. between [and]. TEXT may be null (i.e. no characters) between brackets viz [ ].

Further square brackets may form part of the text providing that each [is matched by a corresponding] (see examples in this section).

When the Macro is called, any parameter in the text will be replaced by characters specified in the actual parameter list. On definition, the text is simply stored, no action is taken on definitions, conditionals, etc., contained within the text.

The following examples make use of the *DEFINE directive:

```
*DEFINE CALL(X)[
11      X
8       X+1]

*DEFINE MST(A,B)[
A
5       B]
```

```
*DEFINE PRINTA[
CALL (QOUT1)
/0      0]

*DEFINE MJUMP(A)[
[MAIN QOUT1]
A
0       +MAIN
/8      0]
```

A Macro may also be defined as having its text on a peripheral. When a call of such a macro is encountered a demand is sent to the appropriate input peripheral and the text may be read in, enclosed between [and] as before. This way different texts may be read in for successive calls of the same Macro. All the usual macro facilities are available in this situation, including nested macros whose texts may be in store, on another peripheral, or on the same peripheral. The directive format is as follows:

*DEFINE Ⓢ MACRONAME (FORMAT PARAMETER LIST)
[PERIPH1]

This macro when called will read text from the paper tape reader.

*DEFINE Ⓢ MACRONAME (FORMAL PARAMETER LIST)
[PERIPH3]

This macro when called will read text from the teleprinter. All other values of PERIPH are illegal. Separators are optional unless shown.

A MACRONAME may be redefined by using a new *DEFINE directive, however this action must not take place within the call of that MACRO. On execution of this directive the original MACRO text and parameters will be deleted from store. A parameter may not be redefined within a call of the macro containing it.

The three MACROS which follow are pre-defined by MASIR; if redefined, this action will not constitute an error.

3.2.1 CALLG (X) Call Global Subroutine

CALLG generates a call of a subroutine which may be in another module. The single parameter replacing X must be declared as a global identifier, the name of a subroutine (See Chapter 8).

Note that the A-register is not preserved on entry to the subroutine. Three instructions are inserted in the object code for each call of CALLG.

3.2.2 MQCHOP

Generates call of QCHOP (Outputs one character to the specified output device; the character is held in internal

code in the A-register). (Each call of MQCHOP generates two instructions.)

3.2.3 QFPCALL

Generates call of QFP - the Floating Point Interpreter in the 900 FORTRAN library (Each call generates two instructions).

3.3 MACRO Calls

Once a Macro has been defined in the text, it can be called at any point by writing its name (MACRONAME).

If a Macro has parameters in its definition, then when called the MACRONAME must be followed by an 'Actual Parameter List' enclosed in parentheses. The actual parameters can be arbitrary characters (excluding commas).

The actual parameters specified may themselves constitute a macro call. In this case the formal parameter will be replaced by the text of the macro called.

The Macro call is replaced by the generated text of the Macro definition, with the actual parameter string replacing the formal parameters and * $ being replaced by an incremented numeric value (See Section 3.4).

Example:

Using the examples defined in Section 3.2.

TEXT INPUT	PROCESSED TEXT
4 VAL MST(2+31, VAL)	4 VAL 2 + 31 5 VAL
PRINTA	11 QOUT1 8 QOUT1+1 /0 0
CALL(SUB)	11 SUB 8 SUB+1
MJUMP (PRINTA)	[MAIN QOUT1] 11 QOUT1 8 QOUT1+1 /0 0 0 +MAIN /8 0

3.4 Numeric Label Generation by * $

Each time a Macro is called, and the pair of characters * $ is contained in the text, the pair is replaced by a set of digits giving an incremental value. The value is set to 1 initially and is incremented by 1 for each call that follows (this facility is useful for generating labels).

The resulting combination of characters formed in this way may be the name of another macro.

Example:

```
 *DEFINE GZ(A,LP)[
 4        A
 9        LL*$
 7        LL*$
 8        LP
LL*$
 ]
 *DEFINE LL3 [SWLIST]
```

The following calls will generate the text shown:

Calls	Text Generated
GZ(D1,D1P) GZ(S1Z,LAB) GZ(&77,SWLIST+2)	4 D1 9 LL1 7 LL1 8 D1P LL1 4 S1Z 9 LL2 7 LL2 8 LAB LL2 4 &77 9 SWLIST 7 SWLIST 8 SWLIST+2 SWLIST

The * S facility may also be used to vary label increments or even vary literals though this is of doubtful value. e.g.

```
*DEFINE MACTST[
4        WS+*$
1        -*$
6        &77*$
5        NUMB*$
8        ;+*$]
```

Would give on a first call

```
4         WS+1
1         -1
6         &771
5         NUMB1
8         ;+1
```

Care should be taken in using this type of construction that the rules governing legal numbers are not infringed. The line 6 &77*$ on the 8th call of the macro will be output as &778 which is an illegal octal number.

The *$ may be combined with a parameter.

```
*DEFINE SUBX (A, B)[
11        A*$
 8        A*$+*$
 B        *$]
```

If called as SUBX (PARAM, /0) gives on a first call.

```
11        PARAM1
 8        PARAM1+1
/0        1
```

In all cases a macro may not be called more than 1023 times without being re-defined. In the case of a name followed by *$, if the total number of characters formed after processing exceeds 6 an error will be given.

The construction LAB*$EL is not legal as the assembler will recognise it as two labels pointing to the same location i.e. LAB1 and EL on a first call.

3.5 Addition to MACRO Text (using * ADDMAC directive)

An existing MACRO may be modified by the use of an *ADDMAC directive , in the form:

ADDMAC (s) MACRONAME [TEXT]

TEXT has the same form as in Section 3.2. The TEXT: is added to the end of the existing Macro text.

Example.

```
*ADDMAC    MST[
9      ERR]
```

Using the example given in 3.2., the call of MST (2 (s) +31, VAL) would now generate.

2	+31	The revised macro definition includes the additional instruction '9 ERR'.
5	VAL	
9	ERR	

It is not possible to define new parameters without completely redefining the MACRONAME, but where there is a direct correspondence between any name in the new text and the formal parameters in the original definitions, the parameter will be replaced in subsequent parameter calls.

If an attempt is made to add text to a macro whose text has been defined as being on a peripheral, an error will be given.

*ADDMAC directives may form part of a macro text e.g.

```
*DEFINE EXTEND[
0       +*$
/4      LABEL

LABEL

*ADDMAC EXTEND[
+0      ]]
```

In this case the macro adds to its own text each time it is called, but it can add to others just as easily.

The example shown gives on a first call.

```
 0      +1
/4      LABEL

LABEL

+0
```

and on a second call

```
 0      +2
/4      LABEL

LABEL

+0
+0
```

When a MACRO name has not been previously defined by a *DEFINE directive but a reference is made to this name by a directive *ADDMAC, it will have the same effect as the *DEFINE directive. (Using the *ADDMAC form, the MACRO cannot have a parameter list.)

Chapter 4: ASSEMBLY FACILITIES

4.1 Conditional Assembly of Text

The directive *IF and *IFNOT are used to assemble text conditionally. These directives are useful for inserting extra facilities during testing.

(a) The *IF Directive

The format for this directive is:

*IF (s) NAMEONE=NAMETWO [TEXT]

After macro processing NAMEONE and NAMETWO should each be reduced to a single identifier, if not, an error will occur. If they are reduced to the same identifier the text will be assembled, otherwise text will be ignored (TEXT may contain any string of characters, with matching square brackets).

Example;

*IF COND=TEST [CALL(PRINT)]

If at the beginning of a program *DEFINE COND [TEST] were defined, then the directive previously stated would cause:

```
11    PRINT
 8    PRINT+1
```

to be assembled.

(b) The *IFNOT Directive

The format for this directive is:

*IFNOT (s) NAMEONE=NAMETWO [TEXT]

This directive is processed as the *IF directive except that the text is assembled if the condition NAMEONE=NAMETWO is not true; otherwise the text is ignored.

(c) Combination of *IF and *IFNOT Directives

The conditional directives *IF and *IFNOT may be combined by the use of & . A full conditional instruction takes the form:

CONDITION [TEXT]

where CONDITION takes any one of the forms:

or

*IFNOT (s) NAMEONE=NAMETWO

or

CONDITION & CONDITION

Example.

*IF COND=TEST1 &*IFNOT STATE=ONLINE
[*DEFINE MONP [15 6144]]

This means that if COND equals TEST1 and STATE does not equal ONLINE, the text will be assembled and so processing of *DEFINE MONP would continue; otherwise the text is ignored.

4.2 The Use of Function Mnemonics

The mnemonics given in the table which follows are used as alternatives to the numeric function codes used in SIR and must not be redefined.

MNEMONIC	MACHINE CODE	FUNCTION
LDB	0	Load B Register
ADD	1	Add to A Register
NEG	2	Negate accumulator and add contents of location
STQ	3	Store Q(Auxiliary) Register
LD	4	Load A Register
ST	5	Store A Register
COL	6	Collate (Logical AND)
JZ	7	Jump if Zero
J	8	Jump
JN	9	Jump if negative
INC	10	Increment (Count)
STS	11	Store S Register (Sequence Control)
MUL	12	Multiply
DIV	13	Divide

MNEMONIC	MACHINE CODE	FUNCTION	
SH	14	Shift	
SHL	14	Shift Left	
SHR	14	Shift Right	
IPO	15	Input/Output	
TER	15 7168	Terminate current program level	
ATB	15 7174	A to B Register	These instructions are available on 905 and 920C only
BTA	15 7175	B to A Register	
ATQ	15 7172	A to Q Register	
QTA	15 7173	Q to A Register	
SKS	15 7169	Skip if standardized	
SKB	15 7170	Skip if B Register is zero after count	
RWG	15 7171	Read word generator	
SRL	15 7176	Set relative addressing	
SAB	15 7177	Set absolute addressing	

See the Technical Manual of the appropriate computer for the exact effects and side effects of each instruction.

Whenever mnemonics have been used as labels in an existing SIR program, then mnemonic detection may be suppressed by *NOMEM (See Chapter 5.10).

If the directive *NOMEM has been used, mnemonic detection may be allowed again by using the directive *MEM.

4.3 Additional Addressing Facilities in MASIR

MASIR provides alternative methods for assembling suitable addresses for programs requiring to communicate between different store modules.

(a) Module Relative Address

Written in the form:-

+ (declared global identifier)

The identifier can be followed by an increment. There must not be a separator between the identifier and the + or the identifier and the increment.

This addressing facility places the module relative address of the declared global identifier in the Current Placing Position (C.P.P.).

The Module Relative Address facility enables a program (A) in one module to communicate with data (D) or program (B) in another module without knowing the module destination of A, B or D.

In this context a module of store is an area of 8192 words of core store, starting at an address which is a multiple of 8192.

Example:

If XL is a declared global data or program label in program B, which is associated with address 200↑ 1 (i.e. 200+8192*1=8392) when unit B is loaded, then:

+XL
+XL+20

are module relative address forms in program A.

If program A is loaded into the module thus:

Module Address	In module 0	In Module 1	In Module 2
+XL generates values	+8392	+200	-7992
+XL+20 generates values	+8412	+220	-7972

Users of the program 903 SIRL 16 may note that the form:

/15 XL

has the same effect as +XL and may be used in MASIR to allow for compatibility with programs written in 903 SIR Language.

(b) Absolute Address

The form used to store an absolute address in the C.P.P. is:

: followed by an identifier (can be followed by an increment).

There must not be any separators between the identifier and the colon or between the identifier and the increment.

Example. (using the example in 3. 8. 1)

:XL would cause +8392 to be stored
:XL+20 would cause +8412 to be stored
:XL-2 would cause +8390 to be stored

independent of the module in which program A was stored.

(c) Module Address Relative and Module Address Absolute

The forms:

+ identifier/ (module address relative)

and : identifier/ (module address absolute)

can be used to store the address of the first location of the module in which the identifier is to be loaded. (There must be no separator between the identifier and +, colon, or /.)

Example.

Using the example of global label XL in program B at 200 ↑ 1 with program A in the modules indicated thus:

Module Address	Module 0	Module 1	Module 2
+XL/ will cause storage of	+8192	+0	-8192
and :XL/ will cause storage of	+8192	+8192	+8192

(d) Literal Address Forms

All the forms described in (a) to (c) can be written as literals (i. e. analogous to literal constants, see Appendix A Chapter 2. 3. 4). They may take any of the forms:

function + identifier
function : identifier
function + identifier/
function : identifier/

In each of these forms a data location is reserved to hold the address form and an instruction is constructed to refer to that data location. (The data location may be

shared between several such references).

Example.

If XL is a global label loader at 200↑1 in program B then in program A:

	MODULE 0	MODULE 1	MODULE 2
0 +XL /8 0 equivalent to	0 +8392 /8 0	0 +200 /8 0	0 -7992 /8 0
or 0 +XL/ /8 XL equivalent to	0 +8192 /8 200	0 +0 /8 200	0 -8192 /8 200

Either of these pairs of instructions will cause a jump to label XL in program B, independent of the modules occupied by program A or B.

```
4     :XL        will cause the value
or               +8392 to be loaded
LD    :XL        into the A Register
```

4.4 MASIR Subroutines

Local subroutines may be written in MASIR in any way convenient to the user. Subroutines which are to be, or might in future be called from outside the program unit in which they are located, should normally be written in the standard form (Since subroutines will normally be called by the CALLG Macro).

The MASIR assembler generates the call CALLG which takes the form of a sequence of code identical to a Fortran subroutine call. See the 900 Fortran Manual for details of this, and of the parameter mechanism necessary if MASIR programs are to call Fortran SUBROUTINES or FUNCTIONS, or vice versa.

Within each store module (block of 8192 words) into which program is loaded, the loader places a set of instructions known as module code. These provide a means of transferring between subroutines in different modules. When the Fortran compiler generates a call of a SUBROUTINE or FUNCTION, it generates a special macro which is processed by the loader. The Macro Assembler MASIR generates the same macro when the source code macro CALLG is used. CALLG is written in the form:-

CALLG(SUB)

where SUB represents the name of the subroutine to be entered, which must be declared as a global label.

The loader macro mentioned above always generates three words of code. If the subroutine in question is loaded into the same module as the calling routine, the loader generates a direct subroutine call, equivalent to the assembly code sequence:-

```
 4   +0
11   SUB
 8 · SUB+1
```

If the subroutine is loaded into a different store module, the loader generates, for each call, 3 words equivalent to the assembly code sequence:-

```
 4   +SUB
11   QMC      (Call SUB via Module Code)
 8   QMC+11
```

where +SUB represents the address of a location holding the address of SUB relative to the calling module.

The module code QMC has the form:-

```
Word

    0;             QMC  +0
 1; to 10;         (Reserved for Fortran, etc. use)
   11;             5 W (Store Relative Address)
   12;             0 W
   13;             6 &760000
   14;             2  QMC
   15;             /5   0  (Store adjusted link)
   16;             6 &760000
   17;             /8   1  (Jump to subroutine entry)
```

This code is automatically duplicated in each module in which program code is stored.

The called subroutine may be written in Assembly code or Fortran. If SUB is written in Assembly code it should have the usual form:-

```
SUB   +0   (Link)
           (Entry point following link)
    .
    .
    .
(Body of Subroutine)
    .
 0   SUB  (Exit)
/8   1
```

If the call of the subroutine is from Fortran, the above example is equivalent to a SUBROUTINE with no explicit formal parameters.

Note that the CALLG and module code mechanism can only be used on the priority level(normally a Base Level (Level 4)). However Subroutine calls on other levels must be handled differently, for example, by a set of macros similar to CALLG, or by avoiding cross module references.

The following macro is used to generate a list of FORTRAN compatible parameter pointers:

QPARAM (parameters)

The MASIR sequence:

```
CALLG(SUB)
QPARAM(parameters)
```

is equivalent to the FORTRAN statement:

```
CALL SUB (parameters)
```

where parameters may be any number of literals, labels or any valid MASIR address field elements, separated by commas. These are laid down in a similar manner to that used by the FORTRAN compiler; each parameter is assembled as /0 n or 0 +LABEL. The second form is used only for global labels not yet located in this unit. See the 905 FORTRAN manual for further details of the method of parameter passing.

NOTE: QPARAM is a special reserved macro which must not be redefined by the user.

Chapter 5: SUMMARY OF MASIR DIRECTIVES

5.1 *DEFINE

The directive *DEFINE stores text specified, with identifier macro name, in the MACRO dictionary.

The format is:

*DEFINE (s) MAC1 (A, B) [TEXT 1]

MAC1 is the MACRO name, if nil text input [ ], if no parameters omit (A, B).

Separators are optional except where marked (s)

5.2 *DELETL

This deletes all MACROS from store previously defined by *DEFINE, but not those specified by *GDEFIN. (The macros CALLG, MQCHOP, QFPCALL and QPARAM are not deleted).

The format is:

*DELETL (s)

5.3 *GDEFIN

This directive is used as *DEFINE except the entries are not deleted by the *DELETL directive.

The format is:

*GDEFIN (s) MAC2 (C, D) [TEXT 2]

MAC2 Macro Name

C, D C and D are parameters

[TEXT 2] Text input

(If no Text input, then write[ ]).

If no parameters included omit ( )

Separators are optional except where marked (s)

5.4 *ADDMAC

This adds additional text to the end of a macro definition; if no macro defined, this directive will act as a *DEFINE directive.

The format is:

*ADDMAC (s) MAC1 [TEXT1a]

5.5 *IF

This directive takes the form:

*IF (s) A = B [TEXTC]

This is a conditional test for the equality of two identifiers; if identifiers are identical, it will assembler text specified.

A and B may be any combination of identifiers or any combination of macro names. The comparison will take place after macro - processing of A or B as required. If after processing, either A or B is not a single name an error will be given.

5.6 *IF NOT

Conditional test for inequality

Format:

*IFNOT (s) C = D [TEXT D]

The same rules apply as for directive *IF, but text will be assembled if the identifiers are not identical.

Note that it is possible to specify a series of conditional tests; text only being output if all conditions are satisfactory.

Example.

*IF (s) A = B &*IFNOT (s) C = D &
E = F [TEXT ABC]

5.7 *LISTLA

This directive sets an Assembler option to list labels on assembly. Each label is listed on the teleprinter followed by the address relative to beginning of program, and G or S (for global or sub-global labels). The format is:

*LISTLA (s)

5.7.1 *NOLIST

This switches off the label listing option. This is the default setting. The format is:

*NOLIST (s)

5.8 *CHECKW

This directive is followed by an octal number and should only be used in special FORTRAN library routines.

When input, the loader program will compare this octal number (xxxxx - indicated in format) with the value generated by the FORTRAN compiler to test for the number of parameters in a FORTRAN call of this unit. If there is no checkword, no action will be taken, When a combined FORTRAN/ ASSEMBLY CODE program is necessitated it may be linked by using the linking loader.

Format is:

*CHECKW (s) xxxxxx

(where xxxxxx represent an octal number).

5.9 *PROG

This directive is followed by a name which is stored (as the program title) in the loader. The name specified will be a global label allocated to the first location of the program unit.

The format is:

*PROG (s) MANEXY

*PROG must precede any instruction or data in a program unit.

5.10 *NOMEM

This directive inhibits the processing of mnemonic functions. The mnemonics for these functions are transferred to the assembler in their original form (like other identifiers) and allow the assembly of a program containing labels which correspond to one or more of the mnemonic function codes.

The format is:

*NOMEM (s)

5.10.1 *MEM

This allows the processing of mnemonic functions. This is the default setting.

The format is:

*MEM (s)

5.11 *CHANGE

The characters

* , ( )

have special significance when used under normal assembly conditions. If MASIR is used as a Macro Generator, when processing FORTRAN tapes, for example, it may be necessary for the assembler to recognise different characters.

Example.

To recognise ↑DEFINE instead of *DEFINE

or 'A : B' instead of (A, B)

where ↑ replaces *
: replaces ,
/ replaces (
\ replaces)

This change in characters is achieved by using the directive *CHANGE followed by four characters which in turn replace * , ( ) respectively.

Example:

Using the previous example the characters * , ( ) become

*CHANGE Ⓢ ↑ : / \

Space separators between characters are optional, but if a line feed is used it will give an error indication.

If it is necessary to reset to the original state, use the form:

↑ CHANGE Ⓢ * , ( )

If fewer than four characters are to be changed, it is still necessary to specify four in the *CHANGE directive.

Example:

*CHANGE Ⓢ £ , ( )

causes £ to be used instead of * to introduce directives, hence:

£CHANGE Ⓢ * , ( )

can then be used to restore to original state.

5.12 *SETDIC

This directive sets sizes of macro and assembler dictionary (standard MASIR assumes a 16,384 word core store). The SETDIC directive may be used to alter this assumption and allocate space to the dictionary areas in which macros and assembler labels etc., are stored.

This directive takes the form:

*SETDIC m n or,

*SETDIC s m n

where s, m, and n are integers, separated by spaces.

m — is the size of macro dictionary required in words of core store.

n — is the size of assembler dictionary required in words of core store (each item requires 4 words of the dictionary).

s — is the size in words of the stack which is a small directory section next to the beginning of the macro dictionary. (Standard setting of the stack is 128 words, and need only be changed by the use of *SETDIC s m n in the unlikely event of a 'stack full' error).

Assuming a standard stack setting of 128 words, if m + n is less than 1600 locations (approx.), both macro and assembler dictionaries will be placed in module Ø, and MASIR will run on an 8K store system. If m + n is greater than 1600 locations, but m is less than 1600 locations, the macro dictionary will be placed in module Ø and the assembler dictionary will be placed from address 8192 to address 8192 + n in module 1.

If m + n > 1600 locations and m > 1600 locations, the macro dictionary will be placed from address 8192 to address 8192 + m, and the assembler dictionary from address 8192 + m to 8192 + m + n.

A macro or assembler dictionary may occupy more than one store module (if store is available). Hence, m or n can be given values > 8192 if required.

The directive *SETDIC should be used directly after the option at the start of a compilation; i. e. before any entries have been made in the dictionary.

5.13 *NOTITLE

This directive suppresses the output of titles to the teletype. Titles will then be ignored in the same way as comments.

This directive also suppresses the output of the end of unit message, normally printed when a % is read.

Format:

*NOTITLE (s)

5.13.1 *TITLE

This directive allows titles to be output to the teletype. This is the default setting.

Format:

*TITLE (s)

Chapter 6: MASIR OPERATING INSTRUCTIONS

6.1 Form of Distribution

The MASIR system is distributed as two tapes: 900 MASIR (Assembler Program) and 900 LOADER.

6.2 Assembler Operating Instructions ('900 MASIR')

Operating instructions for the paper tape version of MASIR are given below. (For operation under the Disc System the appropriate operating system description should be used).

(1) Load the tape '900 MASIR' by initial instructions.

(2) Run out blanks on the punch, ensure that output and input select switches are on AUTO.

(3) Enter at 16, to which the program should reply with ← character.

(4) Type in option in the form:

O n

(letter O followed by a digit (n) in the range 0 to 4). See 6.3 for further details in options.

(5) Load the first tape to be assembled and the type M to continue.

(6) If a halt code is encountered, type C to allow the assembly of a further tape.

(7) When % is encountered, the tape will stop. Run out paper tape and return to step 4 to assemble further programs.

(8) When one or more programs have been assembled, tear off the tape and rewind, backwards i. e. the part first output from the punch should be wound onto the centre of the tape.

6.3 List of Assembler Options

The permitted option values and effects are:

OPTION VALUE	EFFECTS
0	Normal assembly. Read source text and output relocatable binary paper tape.
1	Check context. As for Option 0, but no relocatable binary is generated, however a list of labels and errors may be generated.
2	Macro Processing only. The source text is expanded by Macro generation and a new source paper tape will be punched, containing the expanded form of each Macro. This may be used to generate Fortran or other texts.
3	Macro processing with context check. The expanded source text is output on the punch, but it is also passed through the whole assembler to check for errors. No relocatable binary generated.
4	Output loader halt sequence. This option should be used before assembly of program or string of programs to be loaded in one sequence. It causes a "loader halt" sequence of characters to be output. In the case of paper tape output this will precede the punched relocatable binary programs and therefore will be at the end of tape when loaded. Use of this option is essential when the tape output is to be loaded with a tape reader which does not stop before the physical end of tape is under the lamp (or other character detector). If Option 4 is used, after punching halt sequence program will be processed as for Option 0.

Note: By including bit 4 in the option (i.e. using values 10-14, the macro dictionary is preserved between program units. This is useful when a large number of macros have been defined and more than one unit is to be assembled.

Labels may be listed by the directive *LISTLA. If Option 1 or 11 is used output may be diverted to the punch thus producing lists more quickly. If errors occur the message is preceded by 10 blanks so that it may easily be detected when output on punch.

Using option 2 or 3 or 12 or 13 output may be diverted to the punch to speed listing. The text produced by option 3 or 13 is not legal input for SIR or MASIR.

6.4 Loader Operating Instructions

6.4.1 Function

The loader constructs a fixed address version of a set of programs generated in relocatable binary by the MASIR Assembler. It may be used to load directly into core store or produce a sumchecked binary program on paper tape.

6.4.2 Distribution

As a sumchecked binary tape called "900 LOADER".

6.4.3 Operating Procedure

Operating instructions for the paper tape version of the loader are given below. (For operation in the Disc system, see the appropriate operating system description.)

(i) Load the tape "900 LOADER" by Initial Instructions.

(ii) Enter at 16. A ← should be displayed.

(iii) Type an option in the form letter O followed by an octal number. See para. 6.6 for a list of available options.

(iv) Load the first relocatable binary tape in the reader and type L to enter the Loader.

(v) If there is no Loader halt sequence on the end of tape the reader will unload at the end of the tape. If a change of option is not required, the next tape may be loaded into the reader and may be input by pressing the READ button. If a change of option is required or the last tape has been loaded, press computer RESET and enter at 16.

(vi) However when ← is output, either return to step (iii) and load a new tape, or type M if all tapes required have been read.

(vii) If there are any unlocated labels (global labels, program names or data labels referenced from programs loaded but not included on any tape actually loaded), the names are printed out, each preceded by *ULW. If no unlocated labels, then GO is displayed on the teleprinter.

(viii) If after output of GO, loading was direct into store the program may now be started by typing M again. If the loading process produced a binary tape this should now be complete, run out and tear off the tape.

(ix) If there were unlocated labels, either return to step (iii) to load further tapes, or type M to run the program disregarding the missing labels (0V will be output to indicate override).

If output was to paper tape and override is used the output will then be completed. If loading was direct to core store the program may be started by typing M again (for the third time).

6.5 Entry to Program

The program will be entered at the first location of the first tape loaded (after a loader entry with option with bit 1 = zero), unless one of the program units is entitled MAIN. If there is a MAIN program this will be entered irrespective of the order of loading tapes.

An octal number may be typed as an option before typing M to enter the program, this will be held in the A register on entry.

It may perhaps be advisable that an assembly code program with several facilities be controlled by the use of the option input, instead of separate entry points (common with early 900 series).

6.6 List of Loader Options

The loader option is typed as an octal number, made up of a combination of bits, which have the following effects.

BIT 1 = 0 = 1	if loader to be initialized if loader not to be initialized
BIT 2 = 0 = 1	if everything read is to be loaded. if library scan (only programs or sub-routines that have been called by a previously loaded program are loaded).
BIT 3 = 0 = 1	if loader is to store program in core. if loader is to output program on paper tape or into backing store.
BIT 4 = 0 = 1	if loader is to store program into backing store. if loader is to output program on paper tape (bit 4 is ignored if bit 3 = 0).
BIT 5 = 0 = 1	if program loaded requires 'built-in' routines, set by previous use of bit 6. if program to be loaded does not use the 'built-in' routines.
BIT 6 = 0 = 1	ignore 'freeze' current dictionary and store layout. This causes the routines loaded so far to be 'built-in' to the loader store. Built-in programs are not removed when an option with bit 1 = 0 is used. Thus they may be re-used in different program without re-loading them.
BIT 7 = 0 = 1	ignore list labels
BIT 8 = 0 = 1	print first last messages suppress first last messages
BIT 9 = 0 = 1	halt after warnings, *CLW *COM continue after warnings.

6.7 Loader Setting for Various Store Sizes

The issued Loader tape contains a Loader Setting Routine (LODSET) which allows users to produce loaders 'tailored' to their own requirements, and their own store configurations. The issued tape is pre-set for use on 16384 words of core store, however, if other store sizes are used, LODSET must be used initially. LODSET is overwritten by operation of the Loader, and is not included on tapes punched out by LODSET.

LODSET provides four facilities - identified by the letters, S, F, D and L, which are entered manually by the operator.

They are:-

S	is used to set up the total store size available to the loaded programs. This may be less than or equal to the actual size, which may include "unusable" modules (any store size up to 131072 may be specified). The store size must be a multiple of 8192
F	is used to indicate the free store in each module of 8192 words. For each module within the store size specified by S, the address of the highest free location (+1) is entered. The store available for loading is assumed to extend from relative address zero of that module up to but not including the given address.
D	is used to output a new sum checked binary (paper) tape of the loader, with the new values (set by F and S as standard values).
L	is used to output a sum checked binary paper tape of the adjusted loader, which will load into a specified module, instead of module 0.

Operating Instructions:

(1) Input the issued 900 LOADER tape by initial instructions, ensuring that Input and Output Select are set to AUTO.

(2) Enter at 8.

(3) Type one of the letters S, F, D or L. If S, F or L is used the letter must be followed by one or more integers. These must be typed preceded by + and terminated by semi-colon.

Address may be input in module relative form (e.g. +200↑1; represents address + 8392). Only digits must appear between the + and semi-colon, other layout characters may be used before the +. If newline is input before the semi-colon the number is cancelled, and may be re-input, starting with +.

(4) If S is typed, follow with an integer giving the store size, e.g.:

```
S
+16384;
```

Return to step (3).

(4) If F is typed, it is followed by a list consisting of pairs of numbers. The first number in each pair should be a module number in the range 0 to 15, followed by an address, (the highest free location +1 in the given module). The address must be in the range +0; to +8192; .

The pair of numbers relating to module number 0 terminates the list; return to step (3).

(5) After typing D, sum checked binary tape of the loader is output, with the new settings.

(6) After typing L, type an integer module number.

A sum checked binary tape of the loader, with new settings, will be output. On input of this tape, the loader will be stored in the module specified.

Example of input

```
S
+24576;
F
+2;
+8192;      (0↑2 to 8191↑2 available)
+1;
+0;         (Module 1 unavailable)
+0;
+8130;      (128 to 8129 available)
D
```

0 to 128 of Module 0 are automatically reserved. 8130 to 8191 of Module 0 must be not available if sum checked binary programs are to be produced by the new loader.

Chapter 7: STORE USED

7.1 MASIR Store Used

The MASIR assembler occupies the following store locations in module 0:-

16 to 18
128 to approx 6600
and 8110 to 8135

The remainder of module 0 up to 8110 is used for dictionaries, unless specified by a *SETDIC directive (the standard version of MASIR uses the whole of Module 1 for Dictionaries). To use MASIR on an 8K 900 Series Machine the *SETDIC directive must be used (see Chapter 5.12).

7.2 Store Used by Loader

The Loader program uses store locations 15 to 19 and 128 to approx. 2700. Store above location 2700 (approx) is used for Dictionary, with 5 locations taken for every global label and distinct increment value.

7.3 Store Used by Loaded Programs

Programs are stored downwards in each module between the freestore limits set by LODSET.

Subsequent programs are loaded into the highest address position in which there is sufficient space. If the program cannot be stored without overwriting the loader or dictionary, then a store full indication is output (Error message 05).

Blank COMMON is allocated space from 128 of module 0 upwards. It may however extend beyond location 8191 into Module 1 and beyond, if the space is not already occupied by a program.

Program and Named COMMON will be allocated from the high addressed end of store downwards. The space of program, local data and named COMMON will be allocated at the start of loading each program unit. Named COMMON blocks must not occupy more store in each program unit than that allocated in the first unit in which they occur, whereas it may be possible to extend Blank COMMON, if store is available.

Program must be placed in a different module if there is insufficient room in the current module. Blank and named COMMON blocks may extend across module boundaries.

The loader holds a record of the highest and lowest addresses of free store in each module, which is adjusted when a program is loaded. (If a program unit containing an absolute patch is loaded, these addresses will not be adjusted. It is the user's responsibility to lay out store and prevent overwriting).

Chapter 8: ERROR INDICATIONS

8.1 MASIR Error Reports

The error report output specifies the 'line of text', thus indicating the position of the error. The following is a list of possible error messages and their causes:

ERROR MESSAGE	CAUSE
CALLED MACRO IN USE	A recursive macro call. i.e. call of a macro within expanded text generated by another call of the same macro.
DEFINED MACRO IN USE	Attempt to define a macro recursively.
MACRO DICTIONARY FULL	The space allocated for storing Macro definition, is full (See *SETDIC directive).
STACK FULL	The stack used by the Macro Generator is full, caused by two complex a sequence of nested Macro calls.
LINE BUFFER FULL	More than 120 characters on a line of text (not including blanks and erases).
ILLEGAL CHARACTER	A character not included in the SIR internal character set.
PARITY	An illegal parity character on the input tape.
CORRUPT PARAMETER LIST	Illegal character occurs between name of macro and the actual parameter list in a macro call.
*$ NUMBER OVERFLOW	The number which is to replace * $ (increment) is $>$ 1023
NAME STARTS WITH x	x = name or identifier. This name 'x' has a first character which is not a letter.

ERROR MESSAGE	CAUSE
ILLEGAL FACILITY	This is output if a directive or combination of characters is not allowed.
>5 LETTERS IN NAME *$	The increment which is to replace *$ and the name total more than 6 character
ILLEGAL TEXT FOR PARAMETER COMPARISON	Illegal text :- = conditional comparison on one side of the = sign.
*$ NO. TOO BIG FOR SPACE GIVEN	If the label preceding *$ is n characters long the number re-placing must not be ≥ (6-n) *10.
MISUSED MNEMONIC	A function mnemonic used in conjunction with *$ in a macro.
*IF OR *IFNOT	Illegal character used after *IF etc .
ILLEGAL CONDITIONAL	Format error in conditional statement.
PARAMETER LIST CONTAINS X	A formal parameter list contains illegal character X
PATCH ERROR	A patch has been used in an illegal position, or has the wrong format.
ILLEGAL *CHANGE	The directive *CHANGE consists of < 4 characters.
ILLEGAL *ADDMAC TEXT NOT IN STORE FOR MACRO	The directive *ADDMAC refers to macro text held on a peripheral.
INCOMPLETE [ ] PAIR OR COUNT OVERFLOW	Count of number of nested brackets fail.
CONTEXT ERROR	Character or element illegal in this context.
GLOBAL LIST ERROR	Illegal item in global list, or attempt to declare a global label as sub-global.
ASSEMBLER ERROR	The Assembler program has been corrupted.

ERROR MESSAGE	CAUSE
*IF OR *IFNOT AND NAME =	Illegal character after = sign in conditional
LABEL DECLARED TWICE	A label has been declared again within the scope of the original declaration.
ASSEMBLER DICTIONARY FULL	The space for the dictionary of labels in the Assembler is full. See *SETDIC directive.
NUMBER TOO LARGE	Number too large for given context (any integer greater than 131072).
EUL	Error unlocated label. A local identifier has not been declared as a label at the end of its block.
ERROR IN ALPHA OR OCTAL GROUP	As stated.
ERROR IN LITERAL	An illegal form of literal address.
NUMBER STARTS WITH .	A fraction must be preceded by + or -
TOO MANY PARAMETERS	There are >127 parameters in macro call or definition.
TOO MANY GLOBALS	More than 1023 global labels in a single program unit.
RELATIVE ADDRESS ERROR	Relative address of magnitude greater than 100
TOO MANY FORWARD REFERENCES	More than 1023 forward references to local or sub-global labels as a circuit at the line of error.
*IF & *IFNOT NAME =	Illegal form after &
*NOT PERMITTED	Various errors of sequence, e.g. changing size of dictionary after the identifier has been inserted.

8.2 Loader Error Reports

8.2.1 Loader Error Messages

These are output in the form:

*LDR eeeeee NAME1 NAME2

where eeeeee is an octal number describing the error (see table)

NAME1 is the name of the current program unit.

NAME2 is the name of the item which caused the error.

Neither name is output for error 0. Only NAME 1 is output for odd numbered errors. Both names are output for even numbered errors.

Error No.	Cause
00	First code number is incorrect. Tape has been loaded backwards, or it may not be an RLB tape.
01	Sum check failure.
02	COMMON block not located.
03	Not enough room for this program unit.
04	Label declared twice.
05	Store full.
06	COMMON block name the same as a subroutine name.
07	Program unit larger than 8K, or patched program would cross module boundary.
10	Second or subsequent definition of named COMMON block is of greater size than first definition.
11	Code number not in dictionary.
12	Unlocated label in patch.
13	Illegal format of RLB.
14	Not enough room for named COMMON block.
15	Forward reference table overflow.

All errors cause loading to stop.

8.2.2 Loader Warning Messages

There are two of these:

*CLW NAME1 NAME2
*COM NAME1 NAME2

Where NAME1 and NAME2 have the same significance as for errors.

*CLW checkwords not equivalent.

*COM second or subsequent definition of named COMMON block is of smaller size than first definition; first definition assumed. (Compare error 10).

After a warning message has been output, loading may be continued by typing C.

When M is typed after all units have been loaded, any unlocated labels are printed thus:

*ULW NAME

on a new line for each label.

905 MASIR EXAMPLE PROGRAM

The two programs which follow will be loaded using the standard loader, so that PROG ONE will be stored in store module 1 and PROG TWO in store module 0.

```
*PROG ONE
*DEFINE STG(X) [
0      +X
/5       0
]
*DEFINE SRCAL (X)
[11      X
  8      X + 1]
*DEFINE TEST (A, B, C)
 [4      A]
*IFNOT B = M
 [6      B
  7      C]
*DEFINE M  [ø]
[NXCHAR HDOVER QCHIN READ]
    READ SRCAL (QCHIN)
           5   HDOVER
    TEST (HDOVER, ø, READ)
           SRCAL (QCHIN)
           STG (NXCHAR)
    TEST (NXCHAR, &77, ; + 0)
           8  ; + 0
    HDOVER  + 0
[QCHIN]
QCHIN    +0  }
          1  }
          1  }  This is the Character
          1  }  Input Routine
          1  }
          1  }

%
```

```
*PROG  TWO
↑6144
[QOUT1 HDOVER NXCHAR]

PUNCH      LD    NXCHAR
           IPO   6144
           LDB   +HDOVER
           /LD   0
           IPO   6144

*ADDMAC    SRCALL[
   /0      0
    0      +HDOVER
   /4      0 ]

SRCALL  (QOUT1)
           TER
           J    ; +0

NXCHAR  +0  ⎫
[QOUT1]     ⎪        Number Output
 QOUT1  +0  ⎬        Routine
            ⎪
   etc.     ⎭
```

Symbolic Input Routine

(Notes attached to the SIR ISSUE 6 tape)

APPENDIX A: SYMBOLIC INPUT ROUTINE (SIR)

Contents

Page

Chapter 1: INTRODUCTION

1.1 Purpose of Routine 1
1.2 Advantages of Programming in SIR 1
1.3 Inter-relation of SIR and Machine Code 1

Chapter 2: ELEMENTS OF THE SIR LANGUAGE

2.1 Identifiers and Labels 3
2.2 Words 4
2.3 Instructions 4
2.3.1 Absolute Addresses 5
2.3.2 Relative Addresses 5
2.3.3 Identified Addresses 6
2.3.4 Literal Addresses 7
2.4 Quasi-Instructions 8
2.5 Constants 9
2.5.1 Integers and Fractions 9
2.5.2 Octal Groups 9
2.5.3 Alphanumeric Groups 10
2.5.4 Pseudo-Instructions 12
2.6 Skips 13
2.6.1 Labelled Skips 13
2.7 Comments 14
2.8 Blocks 14
2.8.1 Global and Sub-Global Identifiers 14
2.8.2 Local Identifiers 15
2.8.3 Block Structure 16

		Page
	2.9 End of Tape and End of Program Symbols	17
	2.9.1 End of Tape Symbol	17
	2.9.2 End of Program Symbol	17
Chapter 3:	SUBROUTINES	
	3.1 Use of Subroutines	19
	3.2 Method of Entry	19
	3.3 Method of Exit	20
Chapter 4:	SPECIAL FACILITIES	
	4.1 Patch and Restore	21
	4.1.1 Patch	21
	4.1.2 Restore	21
	4.2 Obeyed Instructions	22
	4.3 Edit	23
Chapter 5:	OPTIONS	
	5.1 Load-and-Go Mode	26
	5.2 Non Load-and-Go Mode	27
	5.3 Check Mode	27
	5.4 Advantages in Use of Non Load-and-Go Assembly for Large Programs	28
	5.5 Summary of Options and Examples	29
Chapter 6:	ASSEMBLY AND LOADING	
	6.1 Assembly of SIR Tapes	31
	6.1.1 Load-and-go Mode	31
	6.1.2 Non Load-and-go Mode	32
	6.1.3 Check Mode	32
	6.2 Loading of Relocatable Binary (RLB) Tapes	32
	6.3 Combination of RLB and Mnemonic Tapes	33
	6.4 Loading Programs into High End of the Store	34
	6.4.1 Loading Programs up to Initial Instruction	34
Chapter 7:	ERROR INDICATIONS	
	7.1 Format of Error Indications and the Effect of Error Indications on Assembly	37
	7.2 Examples of Error Indications in Assembly	37
	7.3 Error Indications given on the Loading of RLB Tapes	38

		Page
Chapter 8:	EXAMPLES OF SIR PROGRAM	39
	8.1 Notes on the Content of Program	40
	8.2 Layout of Program	41
Chapter 9:	STORE REQUIREMENTS	43
Chapter 10:	SUMMARY OF ENTRY POINTS	45
Chapter 11:	GLOSSARY OF TERMS	47

Chapter 1: INTRODUCTION

1.1 Purpose of Routine

The 903 Symbolic Input Routine SIR has been devised to enable programs to be written using a language, the flexible design of which, provides the power of a modified machine code whilst retaining many of the advantages associated with programming in a higher level language.

SIR is an appendix to the full MASIR language defined in Volume 2.1.1.

1.2 Advantages of Programming in SIR

The advantages in use of the modified 903 machine code form of SIR over the use of pure 903 machine code are:

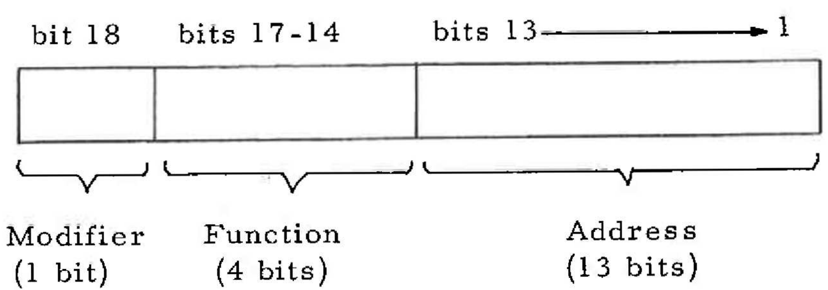
(a) The programmer can invent names to refer to store locations used in a program instead of having to specify absolute addresses for locations so used. The SIR Assembler will recognize names so invented and allocate specific store locations for them.

(b) The programmer can write an instruction using a constant in the address part without having to specify where that constant is stored.

(c) Programs written in SIR can be assembled by the SIR Assembler in any of three different modes (load-and-go, non load-and-go or check mode) to suit specific requirements (Section 6.1.1 to 6.1.3).

1.3 Inter-relation of SIR and Machine Code

In machine code the format of an instruction is as follows:



That is, a machine code instruction consists of 18 bits. For example,

1	0100	0000010000000

This indicates the Modifier bit is set, the function $(4)_{10}$ load accumulator is to be executed and the accumulator is to be loaded with the contents of address $(128)_{10}$ modified by the contents of the B register (modifier bit set).

In this form the equivalent SIR instruction tapes will bear a one-to-one relationship with the machine code instruction tape. The format of the SIR equivalent corresponds to the machine code in that:

A / (solidus) will represent the modifier being set; if omitted the modifier is not set.

The function part is represented by a number in the range 0 - 15.

The address part is represented by an address in any of the formats permitted in the SIR language, i.e. absolute, relative, identified or literal (Section 2.3.1 to 2.3.4).

Hence the equivalent SIR instruction could be written:

/4 128

As it is not necessary (as a general rule) for the programmer to know the absolute address of the data locations within a program, the numeric address $(128)_{10}$ can be replaced by one of the other address forms thus:

4 XDATA or

4 +2

when the SIR Assembler will allocate an address to XDATA or the constant +2. Words in SIR can be written in either instruction or constant form (Section 2.2).

Chapter 2: ELEMENTS OF THE SIR LANGUAGE

2.1 Identifiers and Labels

An Identifier is a name invented by the programmer as a substitute for an address. Identifiers can be devised using any combination of the alphabetic characters and the digital characters 0-9, but the first character of any identifier must be an alphabetic character. For example:

A HOUR TT61 MULT3S	are all valid identifiers

whereas:

32BIT	first character not alphabetic characters	are invalid identifiers for the reasons stated
B(LEN)6	parentheses not alphabetic or digital characters	
T 52	space between T and 52 not permissible	
B-LINE	hyphen not alphabetic or digital characters	

Identifiers are distinguished from each other by their first six characters only. Thus no distinction exists between FLIGHT1, FLIGHT2 and FLIGHTPATH.

Although a distinction is not made in the use of upper and lower case alphabetic characters (FLIGHT, flight, fLighT and FLight being treated as identical identifiers), however it is suggested that either all upper case or lower case characters in an identifier be used; similarly for all identifiers within a program.

Identifiers are 'declared' by being used as labels.

Hence, every identifier must be used as a label once only within its range of validity, i.e. within a program, labels are identified in a unique manner.

Any character other than the alphabetic characters A to Z and the digital characters 0-9 can be used to separate identifiers.

A label refers to the store location into which the word following that label is to be assembled. Labels are followed by one or more separators. For example:

```
OUTPUT    15  6144
  /       └────┬────┘
Label     Word in instruction form (See Section 2.2).

AREA      -
  /       └────┬────┘
Label     Word in constant form (See Section 2.2).
```

Within a program, several labels (separately identified) may be used to refer to a single store location. (Each label will be followed by one or more separators). The labels need not be on the same line as the word following and can appear on more than one line. For example:

8 REPEAT	instruction
BEGIN GO START	labels
ENTRY	label
4 FLAG	instruction

Assuming that the first instruction is assembled in location 2300, then the labels BEGIN GO START and ENTRY all refer to location 2301, into which the instruction 4 FLAG is to be assembled. Similarly, if the first instruction was assembled in location 2336, the labels would refer to location 2337 into which the final instruction would be assembled.

Labels may also be used to specify an absolute address (See Section 2.3.1). In these instances labels are written thus:

CONTINUE = 9

and location 9 can henceforward be referred to as CONTINUE.

2.2 Words

Words are the basic elements of a SIR program. They can be written in two forms, constants or instructions. For example:

+ 304 and - .2667	} are constants

whereas:

15 2048 and /2 CAT+10	} are instructions

On assembly a SIR word (which must be followed by new line character and commence on a separate line) will occupy one store location within the core store. Words are entered into consecutive store locations in the order that they appear in a program, unless the Assembler receives a directive (e.g. patch, skip, option, etc.) to the contrary.

2.3 Instructions

Words written in the form of instructions are introduced by a / (solidus) character or a digit. Each word consists of a function part and an address part; both parts being separated by one or more separator characters (e.g. space).

If the solidus precedes the function part this indicates that the address part is to be modified by the contents of the B register. The function part consists of a decimal integer in the range 0 to 15; each integer represents a 900 machine function (e. g. 4 represents the function load the accumulator). The address part can be written as Absolute, Relative, Literal, or Identified (defined in 2. 3. 1 to 2. 3. 4). An address is assembled as an integer in the range 0 to 8191 and is interpreted at run time as being relative to the start of the store module in which the instruction is placed.

NOTE: References to locations in other store modules are made by B register modified - instructions.

2. 3. 1 Absolute Addresses

An absolute address consists of an unsigned integer not greater than $(8191)_{10}$ and refers to the core store location with that integer as its address. In machine code functions and 15 the absolute address provides further specification of the function using standard conventions.

Examples of absolute addresses are:

4 8180 Meaning, load the accumulator with the contents of location 8180.

15 6144 Meaning, punch the least significant 8 bits the accumulator.

2. 3. 2 Relative Addresses

A relative address can be one of two types, a 'location' relative address or a 'block' relative address. Integers used in relative addresses must be in the range 0 to $(8191)_{10}$.

A location relative address consists of a semicolon followed by a signed integer and refers to a location, the address of which is:

The address in which the current instruction is being assembled + the signed integer value.

Examples of location relative addresses are:

7 ;+3 Meaning, jump three locations forward if zero.

5 ;-1 Meaning, store in the previous location.

8 ;+0 Meaning, perform a dynamic stop.

NOTE: 8 ;0 is an invalid instruction as the integer following the semicolon is unsigned.

A block relative address consists of an unsigned integer not greater than 8191 followed by a semicolon a location with an address equal to:

The value of the unsigned integer + the address of the first location in the current block.

Examples of the use of block relative addresses follow.

		[ONE TWO]	Two global addresses
START		+1 +2	Constants
	4	0;	Load the accumulator with the contents of location (0 + START) = + 1
	5	ONE	Store contents of accumulator in ONE
	4	1;	Load the accumulator with the contents of location (1 + START) = + 2
	5	TWO	Store contents of accumulator in TWO

2.3.3 Identified Addresses

An identified address (introduced by a letter) consists of either an identifier or an identifier followed by a signed integer.

The assembler will replace the identified address with the sum of the absolute address of the location (labelled in a unique manner by the identifier) and the signed integer (called an increment - even if negative in value). The increment must be in the range +4095 to -4096, and the address formed by identifier + or - increment must lie in the same store module as the identifier.

An identified address can be used in the TEXT prior to the declaration of the identifier to which it refers (i.e. prior to the identifier appearing as a label).

An example of the use of an identified address is:

COUNT	+0 +1 +10	3 constants
	4 COUNT	Load accumulator with contents of location COUNT
	2 COUNT+1	Negate accumulator and add content of COUNT + 1 = 10
SELF	9 SELF+3	Jump (if negative) to SELF+3
	10 COUNT	If not, add 1 to COUNT
	8 SELF-2	Jump to SELF-2
	8 SELF+3	Dynamic stop (i.e. JUMP from SELF+3 to SELF+3)

In this example 8 SELF - 2 refers to 4 COUNT.

In 2 COUNT+1 and 8 SELF-2 (which are both incremented instructions) the increments are + 1 and - 2 respectively).

Although an incremented identifier may be referred to prior to its declaration, such references increase the amount of workspace required by the SIR Assembler. Hence, if program necessitates a block of global work space, this should be declared early in the program and any necessary arrays of local work space should be declared near to the start of the block in which they occur. These points are illustrated in the example which follows:

	[MXMULT]		Global Address
	8	MXMULT	Jump to MXMULT
MATRIX		>+400	Reserve store (400 locations) for an array MATRIX - MATRIX declared constant
		+0	
	4	W4S	Coded instruction
		....	Further instructions
	4	MATRIX +265	Reference to MATRIX array previously declared
		....	Further instructions

In this example if 8 MXMULT is assembled in location 3072, then:

4 MATRIX+265 is assembled as 4 3338 (3338 = 3072 + 1 + 265)

NOTE: Skips are detailed in Section 2.6.

2.3.4 Literal Addresses

Literal addresses are introduced by any one of the following symbols:

+, -, =, &, or £.

They are used to ease writing instructions which operate on constants. Their function is indicated in the examples which follow:

Example 1

```
TEN     +10
        ..........
        4 TEN
```

In this instance the identifier which labels a constant to be used at some other point in the program is placed in the address part of the instruction. Whereas:

Example 2

4 +10

In this example, the constant is placed into the address part of the instruction. During assembly, the assembler on reading the end of program symbol % (See also Section 2. 9) allocates a store location to the constant, places the constant therein and finally inserts the address of this location in all the instructions using this constant.

There are four types of literals corresponding to the four types of constants available to SIR. These literals are:

Integers and fractions
Octal Groups
Alphanumeric groups

All of which have the same format as their corresponding constant, for example,

4	-. 2667 Fraction	6	&7777 Octal
2	+360 Integer	4	£E5↑ Alphanumeric

and finally the literal type:

Quasi-instructions (detailed in the next section 2. 4).

NOTE: Literal addresses may only be used with functions 0,1,2,4,6,12 and 13. Any attempt to use other functions will give rise to an error condition denoted by the error message EL.

2. 4 Quasi - instructions

These literals are similar to their corresponding pseudo-instruction constants (Section 2. 5. 4) but differ from them in the following manner:

(1) Every quasi-instruction is introduced by the symbol = which immediately precedes the function bits or solidus (indicating B register modification when present).

(2) The address part of a quasi-instruction must be in absolute form (relative, identified or literal addresses are signified as errors by the error message EO).

Examples of Quasi-instructions are:

4	=8	0	Load accumulator with the constant $2^{16} = 65536$
6	=15	8191	Collate the accumulator with 131071 (binary 1 less than 2^{17})

2.5 Constants

Four types of constants are available in SIR. They are:

Integers and Fractions
Octal Groups
Alphanumeric Groups
Pseudo-instructions

All constants must be followed by a separator character.

2.5.1 Integers and Fractions

These are introduced by a + or - sign. If the + or - sign is immediately followed by an integer then the constant is stored as a binary integer. Viz.

+ 14 is stored as 000 000 000 000 001 110

- 64 is stored as 111 111 111 111 000 000

Integers must be in the range - 131,071 to + 131,071 inclusive. The integer -131,072 may be written as the pseudo-instruction /0 0 or as the octal group &400000.

If a + or - sign is immediately followed by a decimal point (·) then one or more digits, the constant is stored as a binary fraction. For example:

+ .375 is stored as 001 100 000 000 000 000

and - .5 is stored as 110 000 000 000 000 000

The fraction -1 can be written in the same manner as the integer -131072. Fractions can contain up to six digits.

2.5.2 Octal Groups

These are introduced by the symbol &. An 18-bit 903 word can be divided into six 3-bit groups, each group being equivalent to a digit from 0 to 7. Thus a constant, can be written as an & symbol followed by a group of 6 octal digits. For example:

&312705 can be written to represent the binary pattern:

011	001	010	111	000	101
3	1	2	7	0	5

Octal groups of less than 6 digits can be written and in this event the digits are right-justified, i. e.

&42 ≡ &000042

2. 5. 3 Alphanumeric Groups

These groups are introduced by a £ symbol which immediately precedes three alphanumeric characters. These characters are packed (from left to right) into a store location in 6-bit SIR internal code. This internal code is as follows:

900 6-bit Internal Code

External Character	6-bit Code (Octal)	External Character	6-bit Code (Octal)
Space	00	0	20
Newline	01	1	21
"	02	2	22
½ or £	03	3	23
$	04	4	24
%	05	5	25
&	06	6	26
´ acute	07	7	27
(	10	8	30
)	11	9	31
*	12	:	32
+	13	;	33
,	14	<	34
-	15	=	35
.	16	>	36
/	17	10	37
` grave	40	Pp	60
Aa	41	Qq	61
Bb	42	Rr	62
Cc	43	Ss	63

External Character	6-bit Code (Octal)	External Character	6-bit Code (Octal)
Dd	44	Tt	64
Ee	45	Uu	65
Ff	46	Vv	66
Gg	47	Ww	67
Hh	50	Xx	70
Ii	51	Yy	71
Jj	52	Zz	72
Kk	53	[	73
Ll	54	£	74
Mm	55	]	75
Nn	56	↑	76
Oo	57	←	77

NOTES: 1. On input no distinction is made between upper and lower case letters. Letters are always output in upper case (i.e. caps).

2. Newline is a compound symbol consisting of the CR and LF characters. The SIR Input Routine ignores carriage return but recognizes line feed as significant.

3. Tab is equivalent to space.

In packing alphanumeric groups all characters in the table given can be stored with the exception of the following:

(a) ↑ and ← are stored as the octal number 01 (i.e. code for newline) and not in their own codes (Octal 76 and 77).

(b) An alphanumeric group is considered complete if a newline is encountered before three characters have been read after the £ symbol; in this event the group is left-justified (i.e. the remaining characters are considered to have the code 0, the code for a space character). Newline is NOT considered as a character within the groups but acts as any other normal separator. Spaces which occur in the three characters following the £ symbol are treated as normal characters.

The prime function of alphanumeric groups is for storing characters which are to be punched out during a program run. It must be noted that this operation can only be performed when the program using this function contains a print routine and a table for conversion from internal to external code.

Examples of Alphanumeric Groups

Group Required	Form in Store	
	Octal	Alphanumeric Equivalent
£MAN	55 41 56	MAN
£↑H←	01 50 01	Newline H newline
£ space = newline	00 35 00	Space = Space

NOTE: The spaces in octal equivalents are for clarity purposes only. They must NOT be punched.

Since alphanumeric groups containing ↑ and ← cause newline code to be stored, if a representation of ↑ and ← is necessary an octal group must be used. Viz.

Group Required	Octal Equivalent (as written)
↑←↑	& 76 77 76
A %	& 41 05 77

2.5.4 Pseudo-instructions

These are identical in format to ordinary instructions, but are used as constants. For example,

/0 0 can be used to represent the integer -131072

Similarly, it is possible to obey constants as instructions; though the intentional use of this effect is NOT recommended. A failure to terminate an instruction sequence with an unconditional jump (for example, the dynamic stop - Section 2.3.2 and 2.3.3) is liable to result in this undesirable effect.

2.6 Skips

A skip signified thus >, indicates that during assembly a specified number of store locations are to remain unaltered before the SIR Assembler continues filling the store with words. The number of store locations to remain unaltered is specified by an optional + sign and an integer. The specification characters immediately follow the > symbol.

Consider the following example of a skip:

	+ 133	Word
	>+15	Skip
4	8180	Word
5	COUNT	Word
	....	Further coding

If in this example the word +133 was entered into location 5000 of the Core Store, the Skip (>+15) indicates that the next word (the instruction 4 8180) is to be assembled in location 5016 and not 5001. 5 COUNT would then be assembled in location 5017 etc.

The prime function of skips is to reserve locations for work space without actually assigning any values to those locations.

2.6.1 Labelled Skips

Locations left unchanged by skips may be labelled in the same manner as locations occupied by words. For example,

	8	ERROR
		>+4
ALPHA		>+10
MATRIX		>+400
BETA		>+10

In this example, if 8 ERROR is assembled in location 4000, ALPHA refers to location 4005, MATRIX to 4015 and BETA to 4415.

NOTES: (a) The last word of the 10-word vector labelled ALPHA is addressed as ALPHA +9. Similar addressing applies to MATRIX and BETA.

(b) Addresses outside the range indicated in (a) can be referred to by incremented instructions. Thus ALPHA+11, MATRIX+1 and BETA-399 are alternative ways of referring to the second location of the array MATRIX. If the length of ALPHA was changed, the increment relative to ALPHA would have to be changed. Similarly if the length of MATRIX was changed the increment relative to BETA would also have to be changed.

2.7 Comments

These are included in a program to make the print out of that program easier to understand.

Any string of characters valid in the internal code between parentheses () is a comment and is ignored by the SIR Assembler. A separator is not necessary after a comment. A comment may be inserted anywhere between element (except in a Global Identifier List) in a program. Comments must NOT split any SIR element. Example of a comment in a SIR program:

```
9    ERROR2       (NUMBER OVERFLOW ERROR INTEGER >1310,71)
4    INT
5    WS2
.................
```

This section of program would be assembled thus:

```
9    ERROR2
4    INT
5    WS2
...........
```

2.8 Blocks

Every SIR program consists of one or more blocks. The start of a block is signified by a Global Identifier List enclosed in brackets [ ]. This part of a block may only contain identifiers, separators or double quotation marks. Global Identifiers and their uses are described in the next sub-section (2.8.1).

A code body follows the Global Identifier List and is terminated either, by the [symbol (signifying the start of the next block) or by the % symbol (signifying the end of the program).

2.8.1 Global and Sub-Global Identifiers

Global Identifiers from the links between the different blocks of a program. They must be listed in the Global Identifiers Lists at the head of:

(a) the block in which they are declared

(b) every other block in which they are to be valid.

One or more separators must follow each identifier in a Global Identifier List; only identifiers, separators and Sub-Global Identifier markers (") may occur between the brackets which enclose the list. When an identifier is included in the Global Identifier Lists of two or more blocks which are assembled together, that identifier refers to a single address (indicated by a label in one of the blocks - namely, the block in which it is declared). An identifier used 'globally' in some blocks may be used 'locally' in any block in which it is not listed as global.

Sub-Global Identifiers are signified by the use of the double quotes " symbol. If on its <u>first</u> occurrence in a Global Identifier List an identifier is preceded by the " symbol, it is treated as sub-global, thereafter the " symbol is optional for that identifier, whereas, a Global Identifier remains in the SIR dictionary after the end of program symbol % has been encountered (thus permitting communication between several programs held jointly in store), Sub-Global Identifiers are removed from the SIR dictionary when % is encountered. The listing of an identifier as Global or Sub-Global is determined by the first Global Identifier List in which it occurs and is valid for a complete program. An identifier cannot be Global in some blocks of a program and Sub-Global in other blocks of that program.

Examples of Global and Sub-Global Identifiers.

[MOUSE "HAMPSTER" LION WOLF]

MOUSE AND WOLF are Global Identifiers
HAMPSTER AND LION are Sub-Global Identifiers.

2.8.2 Local Identifiers

Identifiers which are neither Global nor Sub-Global are termed Local and have no meaning outside the block in which they are declared.

The same name can be used to represent a Global or Sub-Global Identifier in some blocks, several different Local Identifiers in other blocks and be undefined elsewhere in a program (See 2.8.3 Block Structure).

As previously stated (See Section 2.1) each Local Identifier is declared by being used once and only once as a label in the block for which it is valid. Similarly each Global or Sub-Global Identifier is declared by being used once only as a label in only one of the blocks (i.e. the block for which it is to be valid).

2.8.3 Example of Block Structure

```
        [ SCHOOL "CLASS YEAR ]
SCHOOL        0   YEAR
             /4   17                    block
FORM              =192                  SCHOOL
              7   CLASS
              8   FORM
        [ CLASS SCHOOL "LEVEL ]
CLASS        10   YEAR
              4   YEAR                  block
              9   LEVEL                 CLASS
              8   SCHOOL
YEAR              +0
        [ LEVEL YEAR CLASS ]
LEVEL         8   CLASS                 block
YEAR              > +100                LEVEL

%

        [ LEVEL YEAR "CLASS ]
LEVEL             =5095                 block
↑ LEVEL       0   YEAR                  LEVEL
             10   17
              8   CLASS
        [ CLASS SCHOOL ]
CLASS        10   PUPIL
              4   PUPIL                 block
              7   SCHOOL                CLASS
END           8   END
                  -5

%
```

Block structure example breakdown:-

(i) Programs are named after the Global or Sub-Global Identifier that labels their first instruction.

(ii) Blocks are named after the Global or Sub-Global Identifier that labels their first instruction.

(iii) SCHOOL is Global in both programs.

(iv) "CLASS is Sub-Global in program SCHOOL and another "CLASS is Sub-Global in program LEVEL.

(v) YEAR is Global in both programs and another YEAR is Local to block "CLASS of SCHOOL.

(vi) FORM is Local to block SCHOOL of SCHOOL.

(vii) LEVEL is Sub-Global in program SCHOOL and another LEVEL is the name of the second program.

(viii) PUPIL and END are Local in block "CLASS of LEVEL.

(ix) A third program could refer to the Global Identifiers SCHOOL, YEAR and LEVEL.

NOTE: The program used in the example of block structure has no particular meaning, but is merely used to identify the different types of labels and their use in a block structure.

2.9 End of Tape and End of Program Symbols

2.9.1 End of Tape Symbol (halt code)

A halt code punched at the beginning of a new line on tape causes the assembler to wait pending continued assembly. Assembly is continued when the next tape is in the reader, by re-entering at CONTINUE (See Chapter 6).

When a program is being developed, a tape may contain several blocks each block being terminated by a halt code and followed by several inches of blank tape.

Halt codes are used:

(i) as a terminator on a program which is punched in parts.

(ii) at the end of a patch.

2.9.2 End of Program Symbol (%)

On reading a % symbol at the beginning of a new line:

(i) the assembler displays a list of undeclared local and sub-global identifiers,

(ii) locates all the literals in the order in which they occur in a program, in the consecutive locations immediately following the program,

(iii) displays a list of undeclared global identifiers followed by a 'FIRST, LAST, NEXT' message; this message indicates the upper and lower limits of the store used by the program just assembled and the next location that can be used for the assembly to continue.

Any other symbols (other than newline) on the same line will be ignored; a line is terminated by a newline symbol.

A % symbol should be placed:

(i) at the end of the last tape of a program in load-and-go mode.

(ii) at the end of each section in a non load-and-go mode program (or MASIR program) which is to be assembled as a separate relocatable binary tape.

It is convenient to end all tapes with a halt code followed by a % symbol either, punched from an on-line teleprinter, or from a separate tape (comprising of the characters newline, %, newline, halt code).

Chapter 3: SUBROUTINES

3.1 Use of Subroutines

Subroutines are a series of instructions which are used several times in the same program or are common to several programs. These subroutines provide various facilities, e.g., input and output and various useful mathematical functions (Tan, Cos, Arctan, etc.).

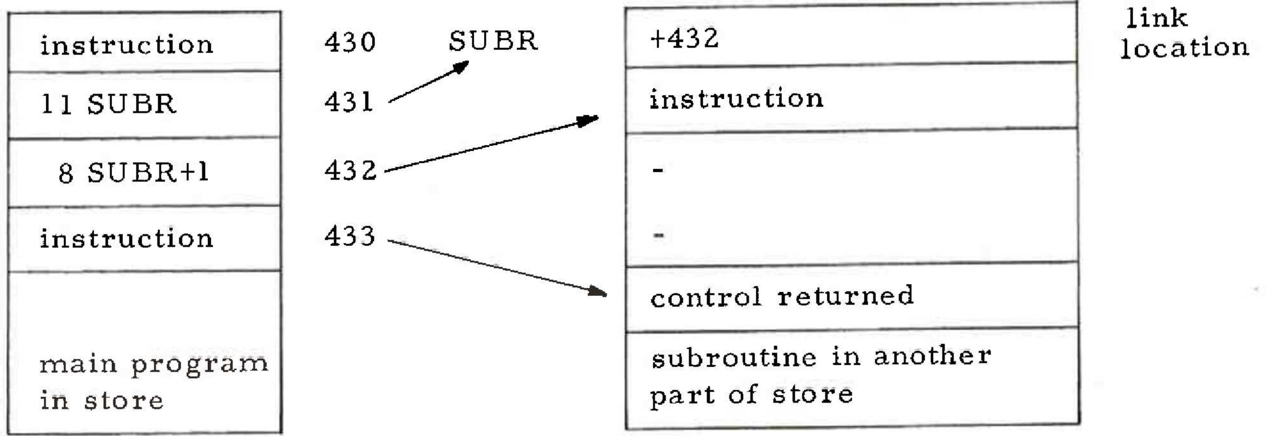
3.2 Method of Entry

To enter a subroutine a special function code is used:

FUNCTION CODE	EFFECT
11	Place the contents of the S register in the location specified.

The S register is an 18-bit register which holds the absolute address of the next instruction to be obeyed. The function 11 actually places bits 13 to 1 of the S register in the location specified, and sets the other bits of the word to zero.

Standard Subroutine Entry:-



The first location in the subroutine is labelled by its name, e.g. SUBR and is known as the <u>Link Location</u>. It should initially be set to a constant +0.

e.g. SUBR +0

In the given example of an entry 11 SUBR sets the link and 8 SUBR+1 jumps to the first instruction in the subroutine.

3.3 Method of Exit

e.g.

201	11 SUBR	SUBR	202
202	8 SUBR+1		- -
203	NEXT INSTRUCTION		- -
		Standard subroutine exit }	0 SUBR /8 1

On completion of the subroutine, exit is initiated by loading the contents of the link location into the B register and completed with the use of a modified jump return control to the main program.

The above method of subroutine entry is adequate if the subroutine is in the same module of 8192 words of core store. If a subroutine is called from a different module of store MASIR should be used. The CALLG macro facility of MASIR enables a subroutine, written with the above convention, to be called from a program in another store module.

Chapter 4: SPECIAL FACILITIES

4.1 Patch and Restore

A patch, is a directive to the SIR Assembler to stop placing instructions, data, program blocks, etc. in current consecutive locations and to place them (consecutively), commencing from a location specified by the patch. At the end of a series of patches compilation of the main program can be continued by the directive restore.

It is important that when using these facilities to ensure that a location, whose contents could be subsequently changed by the SIR Assembler, remains unaltered (These locations contain in their address part information used by SIR, changing of this information could cause corruption of other parts of the program)

4.1.1 Patch

A PATCH is written

↑ A

where A is a constant or any currently located address. Its effect can be defined as

if CPAR = -1 then CPAR:=CPA
then or otherwise CPA:=A
where

CPA is the Current Placing Address, i.e. the address in which SIR will place the next item and

CPAR is a location used to hold a copy of the CPA when inside a Patch (CPAR is initially set to -1 by the Assembler). An example of a PATCH is:

↑ 2048

NOTE: In non-load-and-go mode a patch may be given the value of anunlocated global label, if neither instructions nor constants have been translated before the patch. Other types of address may be used without restriction. (A global label may be unlocated when the Relocatable Binary Tape is made, but must be located when the tape is loaded into tape)

In MASIR the use of patches is restricted (see MASIR Appendix B).

4.1.2 Restore

The symbol $ written by itself on a new line causes assembly to continue from the location which would have been used but for the intervention of a Patch or Patches. Its effect can be defined as:

if CPAR≠ -1 then CPA:=CPAR
then or otherwise CPAR:= -1

Note that the restore facility is not available in MASIR.

Example

```
 4   +200
↑ 2048
 8   L
↑600
 8   ;+0
 $
 5   300
```

If 4 +200 is assembled into location 60, then 5 300 will be assembled into location 61.

4.2 Obeyed Instructions

An instruction written between accents acute and grave (´ `) is obeyed immediately; it is neither stored nor (in the Non-Load-and-Go mode) is it output.

Obeyed instructions use three pseudo-registers:

A pseudo accumulator
A pseudo Q-register
A pseudo B-register

These registers are only affected by obeyed instructions. Examples of obeyed instructions are:

```
´0     COUNT `
´/4    WS `
´5     3000 `
```

In load-and-go mode, a program may be entered at a global identifier (e.g. START) by entering the compiler at 9 and typing or reading ´8 START`.

Obeyed instructions must not be used in SIR non-load-and-go mode.

NOTE: At least one separator must be inserted between the end of the instruction and the terminating grave accent.

Note that obeyed instructions are not available in MASIR. (See MASIR Appendix B).

4.3 Edit

Corrections to mnemonic programs are effected using the library program EDIT. (see Vol. 2.3.2).

Chapter 5: OPTIONS

Options are a form of directive which alter the manner in which the SIR Assembler operates (Note that options are not used in this form with MASIR). They are introduced by an asterisk (*) and followed by an optional + sign and integer. Viz:

*+n

The option is placed at the head of a program, on a new line, before the global label list. The last seven bits of the integer are examined and variations are made in the operation of the assembler as follows:

Bit	Meaning if bit has the value		Availability		
	1	0	Load-and-Go	Non-Load-and-Go	Check
1	Display labels	don't display labels	0 or 1	0 or 1	0 or 1
2	load and go	non load & go	1	0	0
4	clear the store	take no action	0 or 1	0	0
8	punch loader	take no action	0	0 or 1	0
16	continue assembling at 32	continue assembling at NEXT	0 or 1	0	0
32	set dictionary below program	set dictionary below assembler	0 or 1	0	0
64	perform checks only	compile program	0	0	1

An option of *+3 is automatically assumed when assembly is commenced by entering at 8 START. Existing optional conditions are automatically cancelled when a new option is read by the assembler. Options which direct the assembler to perform continuous operations are enforced by using bits 1, 2 and 64; options to direct the assembler to perform a single operation are enforced by using bits 4, 8, 16 and 32.

It is not possible to enforce all combinations of the options indicated by the six bits.

A test is made initially to ascertain whether the assembly is operating in the Load-and-go or non load-and-go mode and so the other bits are examined (as appropriate) to identify the option required.

The differences between the Load-and-go and non load-and-go programs, and the action of the binary loader, are shown more fully in Chapter 6.

5.1 Load-and-Go Mode

When the 2-bit in an option has the value one, the assembler operates in the Load-and-Go mode, i.e. it assembles the source program in the compiler store ready for triggering. A loader cannot be punched when the assembler is in the Load-and-go mode but all the other options are available. Bits are examined in the following order:

(i) 16 bit (continue at 21)

If the 16 bit = 1, succeeding words will be assembled into locations, starting at location 32, until altered by another directive or patch.

(ii) 4 bit (clear the store)

If the 4 bit = 1, the assembler clears all locations between the next vacant location available for assembly to a location first prior to the start location for the SIR Assembler.

(iii) 32 bit (set dictionary below program)

The dictionary is the area of store where the assembler lists all the identifiers and literals it finds within a specific program. It is normally stored just below the store area that holds the SIR Assembler, but if bit 32 =1, it is stored downwards from the location preceding that in which the assembler will place the next word. This option may be used when storing a program in the high end of the store (see 6.4). It is unusual for this option to be read first, since the CPA has to be incremented by a suitable value to allow for the downward building of the dictionary (See example in 5.5). The 32 bit also has a continuous effect of suppressing the checks which normally prevent code being stored at a higher address than the start of the loader.

(iv) 1 bit (Display Labels)

If the 1 bit = 1, when the Assembler finds a label it is displayed (i.e. output to the teleprinter) on a new line with the decimal address of the label to which the label refers. G and S are displayed after the addresses of global and sub-global labels respectively. An extra "new line" is punched when a new block is found.

NOTE: If any error indications occur, they will appear among the output of the labels.

5.2 Non Load-and-Go Mode

When the 2-bit has the value zero, programs are assembled in the non-load-and-go mode (i.e. they are not assembled in the store but are punched out in a special binary loader code) and can be entered into the store by means of a binary loader tape. Tapes produced by this method are called Relocatable Binary (RLB) tapes.

The only options available in this mode are 'punch loader' and 'display labels'. Viz.

(i) 8 bit (Punch Loader)

If the 8 bit = 1 a binary loader is punched in front of the binary tape of the SIR program. This loader enables the tape produced to be loaded by initial instructions.

(ii) 1 bit (Display Labels)

This can only be used on a machine with a teleprinter and a high speed tape punch, and has the same effect as in the load-and-go mode. The Teleprinter-Auto-Tape switch must be set to Auto otherwise the labels will be mixed with the binary output. The address assigned to each label is relative to the beginning of the program, unless absolute patches are used.

5.3 Check Mode

When bit 64 = 1 and bit 2 is set to zero a program will be scanned for errors without being assembled.

The only option available in this mode is display labels. Viz.

(i) 2 bit (Main mode indicator)

This bit must be zero.

(ii) 1 bit (Display labels)

This bit has the same effect as in the load-and-go mode, but may be used in a system which does not have an on-line teleprinter.

5.4 Advantages of Non-Load-and-Go Assembly for Large Programs

The assembly of small programs in load-and-go mode is more convenient than in the non-load-and-go mode; nevertheless there are several advantages to be gained using the latter method of assembly. They are:

(i) RLB tapes are much smaller than SIR tapes, and are read in at a much higher speed.

(ii) Larger programs can be entered using non-load-and-go assembly because:

(a) the program is not stored in the computer and hence the whole store (apart from the area occupied by the assembler) is available to the dictionary.

(b) during loading of the RLB tape, dictionary space is not required for local identifiers and literals; these will have been eliminated during production of the RLB tape.

(c) the loader occupies substantially less store area than the complete assembler. More store is therefore available for the program and the dictionary.

(iii) Large programs may be assembled as a set of program units. If an error is found in one unit it will be much quicker to re-assemble that one unit, and then reload the RLB tapes.

To assemble the largest possible programs it is therefore necessary to operate in the non load-and-go mode, and to keep dictionary space required during loading to a minimum. This can be achieved by:

(i) Avoiding where possible global identifiers, e.g. by reducing the number of blocks, or using sub-global identifiers.

(ii) Avoiding increments to global identifiers which have not already appeared as labels; dictionary space is required for an increment.

5.5 Summary and Examples of Options

Load-and-Go	Translate to paper tape	Check	
2	0	64	Basic mode
1	1	1	Display Labels
4	-	-	Clear Store
-	8	-	Punch Loader
16	-	-	Start placing program at 21
32	-	-	Set Dictionary below program

Add together the numbers in the appropriate column and precede the sum by an asterisk, e. g.

*+19	Load-and-Go, start placing program at location 21, displays labels
*+0	Translate to paper tape
*+65	Checking mode display labels
*+22 >2016 *34 >+2048	Load-and-go, clear store from location 21 to that immediately before the assembly; place the diction in locations 2047 downwards (2047 = 2016 + 32 - 1) and place the program in locations 4096 onwards (4096 = 2016 +32 +2048).

Chapter 6: ASSEMBLY AND LOADING

6.1 Assembly of SIR Tapes

The SIR Assembler is read in by the initial instructions. All tapes written in SIR code can then be read in by entering the assembler at one of the following starting addresses:

Address	Name	Effect
8	START	Cancel all existing dictionaries and begin assembly
9	CONTINUE	Assemble, maintaining current dictionaries

6.1.1 Load-and-Go Mode

In this mode, programs are assembled in the store ready for immediate running. Error indications and (if required by the options) a label list are displayed during assembly.

When a % symbol is read the assembler locates literals, and displays a list of unlocated identifiers followed by:

```
FIRST   LAST   NEXT
 A1      A2     A3
```

where A1 is the lowest address and A2 is the highest address into which program words or data have been stored since entry was made at 8. A3 is the next address into which the program will be stored if assembly is continued.

Example:

```
           *19
X          >10
       4   X+2
       5   X
       8   ;+0
       %         (causes literal +2 to be stored)
```

would cause output on teleprinter as follows:

```
X      32
FIRST  LAST  NEXT
 42     45    46
```

If another tape were loaded after the %, the first word on this tape will be placed in location 46.

NOTE that START entry (8) does not reset the CPA, therefore all programs should normally be preceded by an option (with bit 16 = 1) or a patch.

6.1.2 Non Load-and-Go Mode

In this mode, programs are output to paper tape in relocatable binary (RLB) form; if a loader is required (by option) preceded by the RLB tape. During loading the assembler will form and store a checksum of the instructions.

When a % symbol is read, the literals used are output followed by fifteen blanks and a loader halt code. Any necessary EU messages for global labels are then displayed (EU messages are explained in Chapter 7).

Any errors detected during assembly, halts the punching of the relocatable binary tape and assembly continues in the checking mode.

Several SIR code tapes separated by halt codes, may be assembled to make one RLB tape. After each halt code, continue at location 9.

Each RLB tape must be completed by the assembler processing a newline % newline combination.

If more than one relocatable binary tape is to be assembled, the assembler must be re-entered at location 8 for each new relocatable tape, even if the tape uses global labels in common with preceding tapes. Every RLB tape must be commenced by starting the assembler at START (8) and reading in an option (*0, *8, *1 or *9). It should be noted that no other option should input in connection with the current RLB tape.

6.1.3 Checking Mode

In this mode, error indications and (if required by the options) a label list only are displayed. The only store used is for holding the dictionaries.

6.2 Loading of Relocatable Binary (RLB) Tapes

The binary loader is read in by initial instructions. If during the reading of the loader a character is output continuously, the loader has either been mis-read or mis-punched. When the loader has been read in, RLB tapes can be entered into store at one of the following starting addresses.

Address	Name	Effect
10	START A	Cancel the current dictionary, clear store and read a relocatable binary tape. Start placing at location 32 unless it begins with a PATCH to a different starting address.
11	START B	Read a relocatable binary tape maintaining the current dictionary.
12	START C	As for 10, but CPA is not reset so that the tape will be loaded following the previous program.

If a loader has been punched at the head of a RLB tape using options, when the loader is read in under initial instructions the RLB tape is automatically read in at START A.

A loader is included in the assembler and is correctly located in store when the assembler is read in. This loader can be used to enter RLB tapes using the entry points given above. However, once entry points 10 or 12 have been used it is not possible to assemble source tapes without reading in the assembler again.

During loading, a list of used global labels and their addresses is displayed. If errors are detected, an error indication is displayed and the loader halts; loading can continue to find further errors by entering at START B. On reading a loader stop code loading stops; the loader displays a list of global identifiers still to be located (each preceded by FU) and then displays a FIRST, LAST, NEXT message as described in 5.1. In this A1 refers to the last entry at START A or at START C. The checksum preceding a loader stop code is compared with the checksum the loader made during loading; these sums should be equal, if not an error message will be output.

Every RLB tape must be terminated with a loader stop code (i.e. the last source tape used in an RLB tape production must end with newline % newline).

6.3 Combination of RLB and Mnemonic Tapes

It is possible to read several mnemonic tapes into the store using the assembler, then using the loader in the assembler to read several RLB tapes in at START B. In this instance all tapes share the same dictionary and can communicate with each other via global identifiers. This facility permits library subroutines to be stored as RLB tapes and to be used by a non-RLB SIR program. Note that the last mnemonic tape must end with new line % new line.

6.4 Loading Programs into High End of the Store

Unless the 16 bit in the options indicates a program is to be stored in location 21 onwards, programs read under load-and-go mode enter store immediately above the last program read. Programs can be directed into a specified part of the store either by a patch at the start of a program, or by the use of the continue at 32 option followed by a skip.

6.4.1 Loading Programs up to Initial Instructions

A special version of the SIR Assembler (stored from location 512 upwards) enables programs to be placed in locations normally occupied by the standard assembler. The mode of operation is the same as for the normal assembler. However any program to be placed above location 512 must be preceded by an option *34 or *35 - to place the dictionary in a suitable position and to avoid output of errors when program is stored above the assembler. Programs may be stored from location 3100 upwards to 8179 by this version.

Care must be taken in placing program and dictionary since there will be no check on overwriting of the assembler or dictionary by program.

Chapter 7: ERROR INDICATIONS

Error indications given during assembly.

The following error indications are output to the teleprinter during the assembly of SIR tapes whenever the appropriate error is detected:-

Error	Meaning	Effect in Load-and-Go Mode
E0:	<u>Instruction Error</u> (i) function >15 (ii) address part of quasi-instruction not absolute.	One store location is left unfilled.
E1:	<u>Contextual Error</u> Any impermissible sequence of characters not giving any other error indication.	One store location is left unfilled.
E2:	<u>Octal or Alphanumeric Error</u> (i) Too many characters in an octal or alphanumeric group. (ii) character in octal group other than digits 0-7.	One store location is left unfilled.
E3:	<u>Label Declared Twice</u> Label found identical to a previous label is block where previous label is still valid.	One store location is left unfilled.
E4:	<u>Error in Global Identifier List</u> An impermissible sequence of characters in a global identifier list.	The program is corrupted in an undefined manner.
E5:	<u>Store Full</u> Program is about to overwrite dictionary, or vice-versa. (This may be the result of a Patch error). (E5 after % has been read means that there is insufficient room to locate all the literals used in the program).	The Compiler waits.
E6:	<u>Number Overflow</u> (i) integer outside range - 131,071 to +131,071 (ii) more than six digits in fraction.	One store location is left unfilled.

Error	Meaning	Effect in Load-and-Go Mode
E7:	<u>Buffer Overflow</u> Over 120 characters in line of text (i. e. too many for read buffer).	One store location is left unfilled.
E8:	<u>Illegal Character</u> (i) Misread or mispunched tape. (ii) character on tape having no meaning in SIR (e. g. @)	One store location is left unfilled. In the line of text output the character is replaced by *.
E9:	<u>Stop Code not first Character on Line</u> Characters other than blanks or erases between 'new line' and stop code.	The Compiler searches for newline and then waits. Compilation can be continued. One store location is left unfilled.
EG:	<u>Global Label Error</u> An attempt has been made to redefine a global label as sub-global.	Compilation continues. This may give spurious EU errors.
EL:	<u>Literal Error</u> A literal has been used with an instruction other than 0, 1, 2, 4, 6, 12 or 13.	One store location is left unfilled.
EP:	<u>Patch Error</u> A patch, or obeyed instruction, refers to an unlocated address.	The Compiler waits. Compilation can be continued. A patch skip, option or obeyed instruction must be read next.
EU:	<u>Unlocated Identifier</u> Identifier has appeared but never as a label. Given at end of block for local identifiers, or on reading new line % new line for global or sub-global identifiers.	Compilation continues

7.1 Format of Error Indications and the Effect of Error Indications on Assembly

There are three types of error indications on assembly and each one is preceded by fifteen blanks.

(i) EU: EU error is displayed on a new line; followed by the identifier which has been detected as unlocated and an address. If this address is 8191 the identifier has appeared only in Global label lists or only with an increment; otherwise, the address is that of the last reference to the non-incremented identifier.
The assembler continues to check the identifiers in the dictionary. All F errors halt the loader.

(ii) E5, E9 and EP: These errors are displayed on a newline followed by a block count (i.e. the number of [s encountered since the last entry at START). Assembly halts but can be restarted at CONTINUE.

(iii) EN (all others) EN's information is displayed as in (ii). Also displayed on the same line is the line in which the error was detected. Assembly continues by examining the next line of text for errors.

In every instance of assembly to paper tape, output of the relocatable binary tape ceases, but error indications (and labels if requested) continue to be displayed.

7.2 Examples of Error Indication in Assembly

E2 16 PRINT 6 & 80000	Error output indicates that 8 occurs in an octal group in block 16. PRINT will label the location skipped by the assembler.
E0 10 152048	Error output indicates missing separator giving rise to an impossible function in block 10.
E8 3 8;*0	Error output indicates Illegal character, (probably mispunched) between ; and 0 (replaced by *).

When a EU error indication is displayed after % has been read, it does not necessarily mean an error; it could mean that a Global label has been referred to in a program that has not yet been loaded.

7.3 Error Indication given on the Loading of RLB Tapes

The following error indications are given during loading of relocatable binary tapes:-

Error Indication	Meaning
FA): FD): Mis-read or mis-punched tape	two different kinds of illegal codes on RLB tape
FC: Label used twice	as for E3
FE: Store overflow	as for E5
FF: Checksum failure	punched checksum does not equal checksum added by loader
FP: Unallocated address error in patch	as for EP
FU: Unallocated label	as for EU

Note that:

(i) FC is displayed when a tape with a label is entered at START B when the same label has occurred in a previous tape of the same program (the presence of two identical labels appearing in the same tape would have been detected as an error during assembly).

(ii) FU indications are displayed when a global identifier occurs in one tape and refers to a label on another tape that has not been entered.

(iii) FC and FU only refer to global identifiers because all local identifiers are eliminated during assembly of the RLB Tape.

Chapter 8: EXAMPLE OF A SIR PROGRAM

The program used in the example adds up the absolute values of ten integers in the block headed 'DATA' and stores the answer in location ANSWER. If the sum becomes too large to store in this single store location the letters OF are output on a new line and =/15 8191 is put in location ANSWER.

The program is assembled by entering SIR at START (8). The data block following the halt code can be put on a separate tape and read in at CONTINUE.

The program can then be triggered at location BEGIN. The data blocks occupy locations 61 to 70 and the literals occupy locations 71 to 76: the first literal being placed in the lowest address.

Example 1. Label list produced by example program

```
BEGIN     32 G
LOOP      36
OF        49
END       56
COUNT     58
SUM       59
ANSWER    60 G
DATA      61 S
FIRST LAST NEXT
 32     76    77
```

Example 2. (SIR PROGRAM EXAMPLE)

```
           *+23
[BEGIN "DATA ANSWER]
BEGIN      4  -10 (ENTRY)
           5  COUNT
           4  +0
           5  SUM
LOOP       0  COUNT
          /4  DATA +10
           9  ;+2
           8  ;+2
           2  +0     }
           1  SUM    }  Is the SUM too
           5  SUM    }  large to store
           9  OF     }  in one location
          10  COUNT
           9  LOOP
           4  SUM
           8  END
```

```
OF          4   +10   (PUNCH LINEFEED)
           15   6144
            4   +207
           15   6144  (PUNCH 0)
            4   +198  (PUNCH F)
           15   6144
            4   =/15  8191
END         5   ANSWER
            8   ;+0
COUNT           +0
SUM             +0
ANSWER          +0

H

[DATA]

DATA            +65
                +12
                -14
                -756
                +602
                -5
                +56
                +1
                +0
                -22

%
```

8.1 Notes on Contents of PROGRAM

(i) option +23 means load-and-go, list labels, clear store and start assembly at 21.

(ii) relative addresses have been used for short jumps and identified addresses for larger jumps.

(iii) the identifiers here perform several roles - LOOP, END and OF denote locations to be jumped to. COUNT and SUM denote workspace.
ANSWER denotes the location holding the result.
BEGIN identifies the trigger address in the label list.

(iv) the integer values punched for characters used include parity. In a long program alpha-numeric groups with a table and print routine would be utilised for this purpose.

(v) The program occupies locations 32-70 and the six literals used (-10, +0, +10, +207, +198 and =/ 15 8191) occupy locations 71 to 76. Location 76 is the location for LAST in the print-out.

(vi) The halt code is situated on new line at the end of the first block following the comment (HALT CODE).

(vii) % is preceded and followed by a new line.

(viii) BEGIN and ANSWER have been declared Global labels so that other programs may refer to them. DATA is not needed outside the program and has therefore been declared Sub-Global.

8.2 Layout of Program

There are no set rules for the layout of a program, as separators can be inserted as required.

It is suggested however, that the types of layout used in the example be adopted. Extra 'newlines' can be inserted to divide the print out into legible portions.

If tape preparation equipment without a TAB facility is used, it is recommended that every line is preceded by at least one space character, except for lines starting with a label. For clarity, labels should be punched starting at the left hand margin, and followed by space and a data word, or newline and an instruction.

If a TAB facility is available, every line except a label line should be preceded by TAB, and every label is followed by TAB, then the data or instruction labelled.

Chapter 9: STORE REQUIREMENTS

The standard version of the SIR Assembler occupies locations 5500 to 8179.

When assembling a program the dictionary occupies store just below location 5500, and extends (unless option bit 32 is set) down towards location 8 (See Chapter 5.1). Every dictionary item, label, literal or increment, occupies 3 words of dictionary.

The SIR relocatable binary loader occupies locations 7000 to 8179. When loading a program the dictionary (of global identifiers) occupies locations below 7000, extending towards the beginning of store. Every global identifier, and every separate valued increment to an unlocated global, occupies 3 words of dictionary.

Chapter 10: SUMMARY OF ENTRY POINTS

Entry Points	Action		Reference
8	START		6.1
9	CONTINUE		6.1
10	START A	to load RLB tapes	6.2
11	START B	to load RLB tapes	6.2
12	START C	to load RLB tapes	6.2

NOTE: Locations 13 to 31 inclusive are reserved for use by library programs. Location 20 is used by the SIR Assembler as a continuation address for re-entry at 9.

Chapter 11: GLOSSARY OF TERMS

In the following glossary a brief explanation of each term is given followed where necessary by a reference to a chapter where a full definition or explanation can be found.

ALPHANUMERIC CHARACTER - any alphabetic or numeric tape character which has a six bit internal code representation (Chapter 2.5.3).

ALPHANUMERIC GROUP - a group of up to three ALPHANUMERIC CHARACTERS (Chapter 2.5.3).

ASSEMBLER - the program which reads and translates programs written in SIR code (Chapter 6.1) known as the SIR Assembler.

BLOCK - the main division of a PROGRAM: It comprises a GLOBAL IDENTIFIER LIST followed by a CODE BODY (Chapter 2.8).

BLOCK RELATIVE ADDRESS (N;) - location N is the address of the current block, when N is an unsigned integer. (The first location of a block is relative location zero) (Chapter 2.3.2).

CODE BODY - the whole of a block apart from the GLOBAL IDENTIFIER LIST. It includes constants, instructions and work-space (Chapter 2.8).

COMMENT - information is inserted into SIR programs but ignored by the ASSEMBLER, is used for clarification purposes in the print-out of the program.

CURRENT PLACING ADDRESS (CPA) - the address where the next word will be placed by SIR (Chapter 4.1).

CURRENT PLACING ADDRESS RESERVE (CPAR) - a location holding a former placing address used in conjunction with the patch and RESTORE facilities (Chapter 4.1).

DECLARATION - the use of an IDENTIFIER as a LABEL.

DICTIONARY - an area of store where the ASSEMBLER keeps a list of IDENTIFIERS, INCREMENTS and LITERALS with references to the locations to which they refer.

DIRECTIVE - a PATCH, RESTORE, SKIP or OPTION. Directives tell the ASSEMBLER how and where it is to store the translated program.

DISPLAY - to output information on the teleprinter. If no on-line tele - printer is fitted such information will be output on the punch.

GLOBAL IDENTIFIER - an IDENTIFIER having the same meaning in several PROGRAMS (Chapter 2.8.2).

GLOBAL IDENTIFIER LIST - the list of GLOBAL and SUB-GLOBAL IDENTIFIERS, valid in the BLOCK it heads, it is enclosed in square brackets and occurs at the head of each BLOCK (Chapter 2.8.1).

HALT CODE - a character punched on a SIR mnemonic tape, at the beginning of a newline, which causes the ASSEMBLER to wait.

IDENTIFIED ADDRESS - an address consisting of an IDENTIFIER alone or an IDENTIFIER followed by an INCREMENT (Chapter 2.3.3).

IDENTIFIER - an invented name used as substitute for an address (Chapter 2.1) or the name of a macro.

INCREMENT - a signed integer following an IDENTIFIER to modify its meaning (Chapter 2.3.3).

LABEL - an item in a program located by an IDENTIFIER or located absolutely by being equated to a numeric address preceding a word and referring to a location containing that word (Chapter 2.1).

LABEL LIST - a list of LABELS together with their addresses which can be DISPLAYED during ASSEMBLY (Chapter 5.2).

LITERAL - a constant appearing as the address part of an instruction (Chapter 2.3.4).

LOAD AND GO - a mode of operation in which a SIR program is assembled into the computer store for immediate use. cf. NON-LOAD-AND-GO (Chapter 5.1., 6.1).

LOADER - a tape read in by the initial instructions. It reads RELOCATABLE BINARY TAPES into the store (Chapters 5.3, 6.1, 6.2).

LOCAL IDENTIFIER - an IDENTIFIER which retains its meaning only inside the block in which it is declared (Chapter 2.8.1).

NEW LINE - is compound symbol and consists of the sequence "carriage return, line feed". The SIR input routine neglects carriage return and recognises line feed as significant. The corresponding output routine produces the sequence "carriage return, line feed, blank."

NON-LOAD-AND-GO - a mode of operation in which a SIR program is translated to a RELOCATABLE BINARY TAPE (Chapters 5.2, 6.1.2).

OBEYED INSTRUCTION - an instruction which is obeyed immediately it is read (Chapter 4.2).

OPTION (*+N) - a DIRECTIVE to the ASSEMBLER which enables the programmer to vary the way the assembler operates (Chapter 5).

PATCH (↑+N) - a DIRECTIVE used to correct or control the placing of SIR program. It instructs the assembler to store program in location N onwards (Chapter 4.1).

PERCENT sign (%) - the end of program symbol. On reading it the ASSEMBLER locates constants and checks for undeclared identifiers (Chapter 2.9.2).

PROGRAM - a sequence of blocks terminated by a PERCENT SIGN.

PSEUDO INSTRUCTION - an instruction not intended to be obeyed. For example it can be used as a constant. It is written in an identical format to that used for other instructions (Chapter 2.5.4).

QUASI-INSTRUCTIONS - a literal address in the form of an instruction (Chapter 2.4).

RELOCATABLE BINARY (RLB) TAPE - a special tape holding a SIR program which is output in NON-LOAD-AND-GO assembly (Chapters 6.1, 6.2).

RESTORE ($) - a DIRECTIVE which cancels the effect of a PATCH or series of PATCHES by restoring the placing address to its original value (Chapter 4.1).

SEPARATOR a space or newline. It is used to separate different SIR elements.

SIR - the name given to the 900 Series Assembler (g.v) basic version for 8K stores.

SIR CODE - the set of characters which have representations in six bit internal code.

SIX-BIT INTERNAL CODE - the code in which the ASSEMBLER stores characters three to a location (see code table, Chapter 2.5.3).

SKIP (>+N) - a DIRECTIVE, normally used to reserve store space, which instructs the assembler to leave the next N store locations unaltered (Chapter 2.6).

SUB-GLOBAL IDENTIFIER - an IDENTIFIER having the same meaning in several BLOCKS (Chapter 2.8.1).

PATCH (↑+N) - a DIRECTIVE used to correct or control the placing of SIR program. It instructs the assembler to store program in location N onwards (Chapter 4.1).

PERCENT sign (%) - the end of program symbol. On reading it the ASSEMBLER locates constants and checks for undeclared identifiers (Chapter 2.9.2).

PROGRAM - a sequence of blocks terminated by a PERCENT SIGN.

PSEUDO INSTRUCTION - an instruction not intended to be obeyed. For example it can be used as a constant. It is written in an identical format to that used for other instructions (Chapter 2.5.4).

QUASI-INSTRUCTIONS - a literal address in the form of an instruction (Chapter 2.4).

RELOCATABLE BINARY (RLB) TAPE - a special tape holding a SIR program which is output in NON-LOAD-AND-GO assembly (Chapters 6.1, 6.2).

RESTORE ($) - a DIRECTIVE which cancels the effect of a PATCH or series of PATCHES by restoring the placing address to its original value (Chapter 4.1).

SEPARATOR a space or newline. It is used to separate different SIR elements.

SIR - the name given to the 900 Series Assembler (g.v) basic version for 8K stores.

SIR CODE - the set of characters which have representations in six bit internal code.

SIX-BIT INTERNAL CODE - the code in which the ASSEMBLER stores characters three to a location (see code table, Chapter 2.5.3).

SKIP (>+N) - a DIRECTIVE, normally used to reserve store space, which instructs the assembler to leave the next N store locations unaltered (Chapter 2.6).

SUB-GLOBAL IDENTIFIER - an IDENTIFIER having the same meaning in several BLOCKS (Chapter 2.8.1).

Appendix B: DIFFERENCES BETWEEN MASIR AND SIR FACILITIES

All 900 SIR facilities are described in Appendix A of this manual. The following facilities are either restricted or are not available in MASIR:-

(a) Obeyed instructions (Chapter 4.2 SIR - Appendix) are not available.

(b) Options (Chapter 5 - Appendix) are not available in MASIR. These are replaced by directives (see Chapter 4 MASIR).

(c) The use of the PATCH facilities (Chapter 4.1.1 SIR) is restricted. A patch to a global label or absolute address may occur only at the beginning of a program unit.

(d) The restore facilities (Chapter 4.1.2 SIR) is not available in MASIR.

(e) The use of instruction format

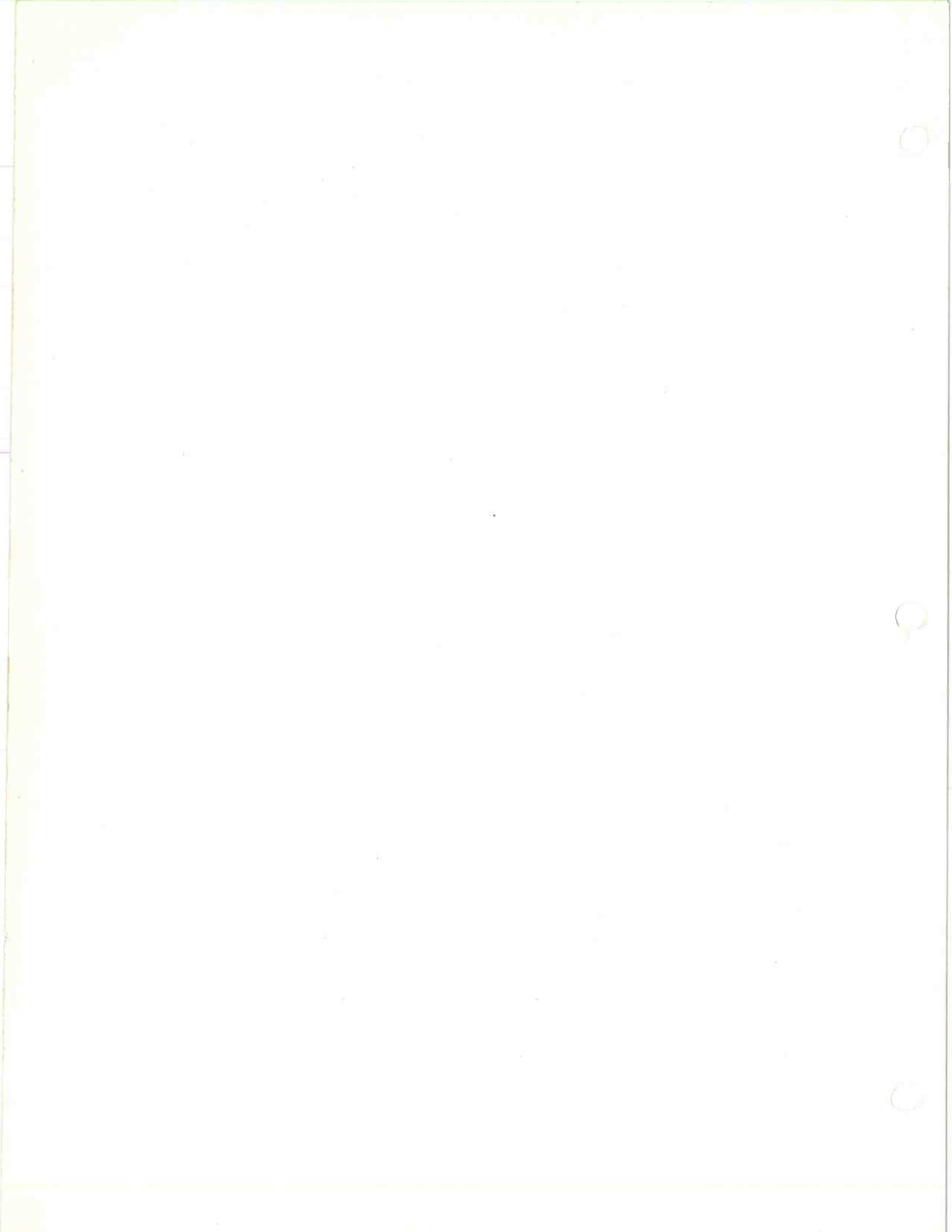
/15 NAME

as defined in SIR Language description is restricted to communication between store modules (where NAME is a global label). See Chapter 4.3(a) of MASIR.

(f) LABEL = 2000 facility is extended to allow the construction of any absolute address. Example:

LABEL = 500 ↑ 2

This label called LABEL is associated with the address 16884 (or 500+8192*2).



Appendix C: MAPLOD (LABEL LISTING PROGRAM)

Function:

MAPLOD is a self contained program used with the 900 RLB loader, to list program labels with addresses assigned to these labels by the loader. It can also list unlocated labels and the references made to these labels in the user's program, and the amount of free store still available in each module. (NOTE. Free store is not meaningful if absolute patches have been used).

Distribution:

MAPLOD is distributed as a sum checked binary tape, suitable for input by Initial Instructions. Two versions are supplied:

(a) A start address at 4∅96 ↑ L; this is the version normally used.

(b) A start address at 256↑ L; this is for use when version (a) would overwrite the loader dictionary or the loaded program. It has the disadvantage that it overwrites part of the loader.

L indicates the store module which contains the loader.

Operating Instructions:

1. Using the 900 loader, load RLB tapes as usual.

2. If output is required on paper tape, set SELECT OUTPUT to PUNCH and run out a blank leader.

3. Load SCB of MAPLOD version (a) or (b) under Initial Instructions. It will be stored in the same module as the 900 loader and will self trigger.

4. Type:

L for located labels and addresses
U for unlocated labels
R for references to unlocated labels

A list of free store is output at the end of each of the above lists.

An asterisk output on a line by itself indicates odd parity on teletype input.

The result is undefined if the R option is used after loading to paper tape or backing store (loader option bit 3 set).

Store Used:

MAPLOD occupies 558 words of store.

Its SCB loader occupies locations 8132 to 8179 inclusive.