# Implementing software using the L language

Revised Second Edition

April 2004

P.J. Brown, R.D. Eager

Copyright © 1968, 1972, 2004 P.J. Brown, R.D. Eager

Permission is granted to copy and/or modify this document for private use only. Machine readable versions must not be placed on public web sites or FTP sites, or otherwise made generally accessible in an electronic form. Instead, please provide a link to the original document on the official ML/I web site (`http://www.ml1.org.uk`).

# Table of Contents

## Preface to the Second Edition

The first edition of this manual was published by the Cambridge University Mathematical Laboratory under the somewhat verbose title *Technical Memorandum 68/1: the use of ML/I in implementing a machine-independent language to bootstrap itself from machine to machine*.

This second edition has been re-published at the University of Kent, and contains minor additions and corrections to the first edition.

Readers should note that there now exists an alternative and usually better way of implementing ML/I, which is described in the manual *Implementing software using the LOWL language*.

The bulk of the text has remained unchanged since the first edition, the following being the only facilities that have been significantly extended: `READ`, `OUTPUTID`, `MDERPR`, `LAYCHAIN` (all concerned with startlines), `SUBROUTINE` (use of a stack for return addresses), `HETABLES` (S-variables) and initialisation (S-variables and `GHSHPT`). Minor textual changes have been avoided where possible. For example semicolon is still used to terminate operation macros (though most implementations now use newline instead), `~` is assumed to be the insert marker (rather than `%` as in later manuals) and `{`*NL*`}` (rather than the layout keyword `NL`) is used to stand for the newline character within structure representations.

There have also been changes to the character set used in listings, paper tapes, etc. due to the differences in the codes used by the `ICL` 4130, where such items are now produced.

## Preface to the Revised Second Edition

This edition has been converted to electronic form using Texinfo, and the opportunity taken to make some minor changes which do not, however, merit a new edition.

The text has been updated to reflect current versions and usage of ML/I. For example, newline is now used to terminate operation macros, and examples of structure representations etc. have been updated to reflect this; also, `%` is assumed to be the insert marker. Source code in L is now distributed in ASCII, and the opportunity has been taken to use more meaningful characters for the 'and' operator (now `&`), the 'or' operator (now `|`), and the character representing newline (now `$`).

Some substantive errors (present even in the first edition) have been corrected, as well as transcription errors introduced in the second edition.

Indexes have also been added.

# 1 The Transfer of ML/I from Machine to Machine

In order to make it easier to transfer ML/I from machine to machine most of the logic of ML/I has been coded in a ‘machine independent’ language which has been specially designed for the purpose. This language is called L and is intended to have the following properties:

a. L is suitable for describing the logic of ML/I.
b. Programs in L are easily readable so that L can serve as a ‘publication language’ in the same way as ALGOL does.
c. L can be mapped by macro replacement using ML/I into the assembly language of any desired machine.
d. L can be mapped by macro replacement using ML/I into any suitable high-level language. To be ‘suitable’ a high-level language must contain, in one form or other, the equivalent of all the facilities of L. In particular it should be capable of performing arithmetic operations, character operations, and operations on blocks of data.

However, certain parts of the logic of ML/I are manifestly machine-dependent, and hence there is no point in trying to describe them in a machine-independent way. Machine-dependent operations include I/O, the hashing function and type conversion routines. Hence the logic of ML/I is divided into two parts as follows:

1. The *MI-logic*. This is the machine-independent part of the logic and is described in the language L.
2. The *MD-logic*. This is the machine-dependent part of the logic, which requires hand-coding for each implementation. It is described verbally in this manual.

In a typical implementation of ML/I the MI-logic might map into 3000 orders and the MD-logic into 500.

## 1.1 Procedure for transfer

The use of the language L enables any existing implementation of ML/I to be used to create an implementation for a new machine. The procedure for doing this is as follows:

a. An *object language* is selected into which L is to be mapped. The object language may be the assembly language of the object machine or it may be a high-level language for which a compiler exists on the object machine.
b. A *base machine* is selected. The base machine must possess an implementation of ML/I but apart from this the choice of base machine is arbitrary.
c. Macros are written to map L into the object language. These are called the *mapping macros*.
d. When the mapping macros have been debugged, they are used to map the description in L of the MI-logic of ML/I into an equivalent description in the object language. This operation is called an *L-map*. L-maps are performed on the base machine.
e. The MD-logic is encoded by hand in the object language. This operation should be carried out in parallel with the writing of the mapping macros since there may be some interaction between the two. The MD-logic should be debugged using the object machine.

f. The object language versions of the MI-logic and MD-logic are combined and assembled (or compiled) on the object machine. When this code has been debugged the result is a working version of ML/I on the object machine. Debugging should not be a lengthy operation since if the mapping macros have been correctly specified there should be no errors in the MI-logic. If however these macros are incorrect to the extend that the generated version of the MI-logic is riddled with errors, it may be necessary to go back to step c) above.

## 1.2 Extensions of the technique

The procedure outlined above can be used not only for ML/I itself but for any piece of software. Assume that it is desired to apply the procedure to another piece of software, which will be called S. Then a new language, M, is designed, which has similar properties to L except that it is for describing the logic of S rather than ML/I. In practice M may be much the same as L but whereas L has special statements for stacking and for following down chains, which are special operations heavily used in the logic of ML/I, M may have statements for performing operations peculiar to S. Typical such operations might be accessing the particular type of dictionary used in S or manipulating the particular type of list structure used in S.

In most cases once mapping macros to convert L into some object language had been written, it would only require a few extra macros and a few deletions of existing macros to derive a system of macros for mapping M into the same object language.

## 1.3 Efficiency of Generated Code

The efficiency of the macro-generated code depends on how much trouble has been taken in writing the mapping macros.

If efficient code is to be generated then these macros must be designed to recognise all sorts of special situations and, as a result, the macros will be larger and will take much longer to debug. However, even if little effort is made to perform optimisation the generated code will not normally be grossly inefficient; the inefficiency will normally be between 5 and 50 per cent. The reason why the generated code should be reasonably efficient is that L has been specially designed for describing the logic of ML/I, and the basic statements in L therefore correspond to the most heavily used operations in ML/I. Hence even if special cases are not optimised, the resultant code should be much better than the code that would result if the logic of ML/I had been described in some predefined high-level language because the facilities offered in the high-level language would be unlikely to correspond to the most heavily used operations of ML/I and so some of these operations would need to be described in a rather clumsy way (for example when using PL/I, algorithms involving string-manipulation can often only be expressed in a logically clumsy way and, even if the PL/I produced highly optimised code, the resultant code would still be much less efficient than would result if there had been special string manipulation operators tailored to the problem in hand).

To summarise, it is better to tailor the software-writing language to the software rather than vice versa.

## 1.4 Advantages of the Technique

Of course it is always possible to code the MI-logic of ML/I by hand for the object machine. This has the advantage that the resultant code should be more efficient, though the implementor would need to have a thorough knowledge of the working of ML/I to gain much in this direction. Against this, the following advantages can be claimed for converting the MI-logic by macro-generation.

a. Macro-generation requires considerably fewer man-hours.
b. Debugging of the generated code should be a very much shorter operation.
c. If a new version of ML/I becomes available it will be a trivial operation to implement it.
d. The technique is particularly advantageous if the object machine is newly manufactured and has no software or if the object machine is on order but not yet available since most of the work of implementing ML/I can be done on another machine.
e. As a by-product, a compiler for L on the object machine is available and can be used for other purposes.

## 1.5 Magnitude of an L-map

It is impossible to give any firm estimate of the number and size of the mapping macros necessary to perform an L-map, since this depends so much on the object language. In general the higher level the object language, the fewer features of L that require complicated mapping macros. For example, arithmetic expressions in L require very little translation to be made into expressions in PL/I whereas extensive translation is necessary to map them into assembly language.

Another variable factor is the degree to which names of variables, labels and subroutines in L require translation. Often no translation is necessary, but FORTRAN, for example, would require the identifiers representing labels to be mapped into numbers.

Hence although each L-map will normally have a mapping macro corresponding to each statement in L, the number of mapping macros needed to deal with sub-components of L will vary considerably between L-maps. However if a figure is required for the time to perform an L-map, experience so far indicates an average time to perform an L-map into an assembly language of about six man weeks plus the time taken for the implementor to learn to use ML/I and the object language, if he does not know them already.

## 1.6 Organisation of this manual

The remainder of this manual is devoted to a description of L and a description of the MD-logic of ML/I. Hints on the writing of an L-map are provided. It is hoped that the discussion will be of interest to three types of reader:

a. Readers who wish to implement ML/I by performing an L-map.
b. Readers who wish to implement ML/I by hand-coding.
c. Readers who are interested in machine-independence.

# 2 Basic Details of L

## 2.1 Character set

The characters used in L consist of the upper case letters `A`–`Z`, the digits `0`–`9` and the following:

```
, = ' [ ] & | ( ) + - / *
```

together with the following characters which occur only within character string literals:

```
; $
```

In addition the layout characters space, tab and newline are used. The only significance of layout characters is:

a. Newline is used to terminate statements.
b. Spaces are significant within character string literals.

Apart from this, layout characters are used redundantly to improve the layout of the logic. Redundant tabs and newlines often appear between statements and redundant spaces often occur adjacently to statement delimiters.

## 2.2 Notation and Terminology

The notation used for describing the syntax of L is the same notation as that used in the ML/I manual except that the brackets `<<` and `>>` are used instead of `[` and `]`, since the latter occur naturally. To illustrate the notation, the syntax of the ML/I `MCGO` statement would be written:

```
MCGO {arg 1} << (IF     ) {arg 2} (=   ) {arg } ? >>;
                (UNLESS)          (GR )
                                  (etc.)
```

Within the description of the semantics of L the notation of ML/I inserts is used, with `%` as the insert marker as in the *ML/I User's Manual.* Hence `%A1.` refers to the first argument, `%A2.` to the second, and so on. `T1` is used to represent the number of arguments and so `%AT1.` refers to the last argument.

In order to aid the writing of an L-map, structure representations are specified for all the elements of L. However in many cases delimiter structures can be written in many possible ways, and hence the structure representations specified should be viewed as suggestions rather than as absolute requirements.

## 2.3 Identifiers

Identifiers are used in L for the names of variables, labels and subroutines. Each identifier is a sequence of between three and six characters, the first of which is a letter and the remainder of which are either letters or digits.

## 2.4 Layout of the Logic

The MI-logic is divided into pieces called *SECTIONs*. These are delimited by the `SECTION` and `ENDSECT` statements in the following way:

```
SECTION name, subsidiary comment
<< statement *>>
ENDSECT name
```

where the name following `ENDSECT` is the same as that following `SECTION`. The name serves no other purpose than to identify the SECTION. The SECTIONs of the logic and their names are as follows:

| | | |
|---|---|---|
| a. | `VARS.` | Variable declarations. |
| b. | `INVALS.` | Initialisation before main logic is entered. |
| c. | `MAIN.` | Main logic. |
| d. | `MAINSUBS.` | Main subroutines. |
| e. | `OPMACS.` | Operation macros. |
| f. | `DEFSUBS.` | Subroutines for setting up definitions. |
| g. | `ERR.` | Error routines. |
| h. | `ENVPR.` | Logic to print out environment. |
| i. | `MACNAMES.` | Operation macro names. |
| j. | `DELS.` | Delimiters and keywords. |

All statements in the `VARS` SECTION are declarative statements, which are described in Chapter 5 [Declarative Statements and Initial Values], page 37. All statements in the `MACNAMES` and `DELS` SECTIONs, which are called collectively the *data SECTIONs* are data-defining statements, which are described in Chapter 6 [The Data SECTIONs], page 40. The remaining SECTIONs are called collectively the *program SECTIONs* and contain only executable statements, which are described in Chapter 4 [Executable Statements], page 17.

The main purpose of SECTIONs is to give some flexibility in the organisation of an L-map. The order of SECTIONs may be changed freely if desired and some SECTIONs may be hand-coded and some may even be totally ignored in some L-maps. Other possible uses of SECTIONs are:

a. They may be mapped into statements which control the format of the object code, e.g. statements which start a new page of listing.
b. They may be useful if the object code needs to be segmented.
c. Since different SECTIONs contain different kinds of statements it may be convenient to map groups of them separately. However, since all the statements have different formats there is no harm in applying all the mapping macros to all the SECTIONs.

There are, in addition to `SECTION` and `ENDSECT`, two other layout statements, namely `PRGSTART` and `PRGEND`. Each of these occurs once in the logic, `PRGSTART` at the very start and `PRGEND` at the very end. Neither statement has any arguments. In most L-maps these statements will either be deleted or be converted into control statements for the object language.

The structure representations of the four layout statements are as follows:

a. `SECTION, NL`
b. `ENDSECT NL`

c. `PRGSTART NL`
d. `PRGEND NL`

## 2.5 Comments

L contains three types of comment, all of which occur freely throughout the logic. The types are:

a. *Headings.* These are major comments, which indicate the beginning of a logically distinct piece of logic. Headings have form:

   ```
   /+ text +/
   ```

   Headings always occupy a line by themselves.

b. *Subsidiary comments.* Subsidiary comments are written:

   ```
   // text //
   ```

   Subsidiary comments of this form occur as arguments to the `SECTION`, `SUBROUTINE` and `BLOCKDEC` statements. As well as this, subsidiary comments can occur by themselves, in which case they occupy a single line, though they may be preceded by tabs.

c. *Statement Prefixes.* Some statements in L which may require special action in certain L-maps are immediately preceded by comments of form:

   ```
   /- text -/
   ```

As an example of a statement prefix, an assignment statement on which arithmetic overflow may occur is written:

```
/- OVP -/ SET ...
```

In an L-map where it was desired to take special action on such an assignment statement,

```
/- OVP -/ SET
```

would be recognised as a macro name. The list of possible statement prefixes is given in Appendix A. However this list may be extended at any time, if some L-map requires certain statements to be identified. In order that new prefixes should not upset existing L-maps, each L-map should contain the skip definition:

```
MCSKIP / WITH - - WITH /
```

to delete all prefixes. This skip would, of course, be overridden for those prefixes it was desired to recognise. For example, if

```
/- OVP -/ SET
```

was a macro name, then this would naturally override the skip.

The delimiter structures for comments are:

a. `/ WITH + + WITH /`
b. `/ WITH - - WITH /`
c. `/ WITH / / WITH /`

Comments should always be defined as skips or straight-scan macros.

## 2.6 Readability of object language listing

A basic decision to be taken on each L-map is whether the generated object language program is to be made to 'look nice'. If not, all comments can be deleted. If so, comments must be mapped into object language comments and a fair amount of trouble must be taken with newlines, tabs and spaces in order that the format of the object code will look reasonable.

## 2.7 Statements and labels

A program in L is similar to a program in most other programming languages; it consists of a sequence of statements, some of which may be labelled. In L a label is written:

```
    [ identifier ]
```

Labels are applied to data-defining statements as well as executable statements. In the former case they are called *data labels* and in the latter case *program labels*.

## 2.8 Use of Storage and Stacks

Several statements in ML/I use the *stacks*. To understand the use of these it is necessary to consider the way ML/I makes use of storage.

When ML/I is loaded into a machine it may be loaded in one contiguous chunk or, provided the loader can take care of the necessary linkage, the MD-logic and the various SECTIONs of the MI-logic can be split off from one another. Some of them might even reside on backing storage. In addition to the storage required for its logic, ML/I requires a fairly large area of contiguous storage for workspace, which is used for its two stacks. These two stacks are the *forwards stack*, which starts at the beginning of workspace and works towards the end, and the *backwards stack*, which starts at the end and works towards the beginning (if the two stacks ever meet, a process is aborted for lack of storage). The two stacks are used to contain macro definitions, etc., and to preserve information in recursive situations.

In a batch processing environment the workspace will normally occupy all the free storage of the machine, but in a multi-programming environment the size of the workspace required, which depends mainly on the number and size of macro definitions, might be specified by the user.

# 3 Components of Statements

This chapter describes the components that make up the arguments of statements in L. These arguments may consist of constants, variables or indirect addresses. In some cases these are combined to form arithmetic expressions. For most L-maps it will not be necessary to convert the form of variable names but macros will be needed for the following purposes:

a. To convert machine-dependent constants.
b. To perform indirect addressing.
c. To perform expression evaluation.

Constants and variables occur in all SECTIONs of the logic, but indirect addresses and expressions only occur in the program SECTIONs.

## 3.1 Data types

The following data types, which apply to constants, variables and indirect addresses, occur in L: pointer, switch, character, number. These data types are described below.

a. *Pointers.* Pointers point at addresses on the stacks and in the data SECTIONs. The following operations may involve pointers: addition, subtraction, assignment, numerical comparison, passing as argument.
b. *Numbers.* Data of type number can take on positive or negative integral values. The absolute value of numerical data cannot exceed the maximum number of characters that can be fitted onto the stacks, except for the variable `INVOCT` (the count of calls) and variables used as line counts or in calculating the values of macro expressions. These latter are potentially infinite. There is no point in extending the precision of numerical data just to cater for these few variables as overflow is unlikely and not important even if it does occur. Most implementations ignore overflow, but, just in case it is desired to detect it on some particular implementation, assignment statements on which overflow is possible have been prefixed:

   ```
   /- OVP -/
   ```

   Numerical data is subject to the same operations as pointer data.
c. *Switches.* Switches can take the numerical value 0, 1, 2, 3, 4, 5, 6 or 7 (the values `TRUE` and `FALSE` are also used, but these are represented as 1 and 0, respectively). Switch variables are subject to the following operations: assignment, and'ing, or'ing, comparison for identity, passing as argument.
d. *Characters.* Character data can represent any character in the set used by the implementation. Character data is only subject to one operation, namely comparison for identity (character data is moved about using the block move statements (see Section 4.3 [Block Moving Statements], page 25), but never character by character).

### 3.1.1 Representation of Data Types

Data types need only be differentiated on an L-map if they are stored in different ways. It makes an L-map very much easier if all data types are represented in the same way. This will normally be convenient on a word-oriented machine, but it would be rather wasteful, for instance on System/360 to represent switches or characters as full words. Failing complete identity of data types, the following equivalences make an L-map easier:

a. Numbers and pointers are represented in the same way.
b. (less important) Switches and characters are represented in the same way.

The first consideration is important as it obviates the need to examine the data types when generating code for arithmetic expressions.

It is very undesirable to represent characters in a 'packed' form in such a way that they occupy less than the basic storage unit of the object machine, since this involves considerable problems with code generation, addressing and alignment.

Examples of how data types have been treated for various object languages are:

a. *PDP-7 Assembly Language.* No differentiation between data types. All data occupies one 18-bit word.
b. *IBM System/360 Assembly Language.* Characters and switches occupy 1 byte (of 8 bits), and numbers and pointers occupy 4 bytes.
c. *PL/I.* All data is represented as:

```
    FIXED(15) BINARY STATIC
```

## 3.2 Variables

Variables in L are represented by identifiers. All variables are represented are declared in the `VARS` SECTION. Variables may be of type pointer, number or switch. There are no character variables. The last two characters of the name of a variable represent its type as follows:

| | |
|---|---|
| `SW` | means switch, e.g. `MASKSW`, `INSW`. |
| `PT` | means pointer, e.g. `SPT`, `PRP2PT`. |
| `CH` | is not used. |

Anything else means number, e.g. `TYPE`, `IDLEN`.

All variables are scalars and have global scope. There is never any need for storage to be assigned to variables dynamically.

## 3.3 Constants

Many constants in L of type number or switch are represented as integers. These integers are always in the range 0–9 except for one case noted in Section 6.2.3 [Hash-Tables and their Definition], page 43.

However in many cases the value or the representation of a constant in L will depend on the object language or on the object machine. These constants are represented in L by *constant-defining* macros, each of which will be replaced by the appropriate value during an L-map. The various constant-defining macros are described in the following Sections.

### 3.3.1 The `OF` and length-defining macros

Before describing the `OF` macro it is necessary to describe the six *length-defining* macros, which act as subsidiary macros to the `OF` macro. The length-defining macros all represent positive integer constants as follows:

1. `LPT` = number of units of storage occupied by a pointer.
2. `LNM` = number of units of storage occupied by a number.
3. `LSW` = number of units of storage occupied by a switch.
4. `LCH` = number of units of storage occupied by a character.
5. `LICH` = `1/LCH`. Special action is necessary if `LCH` is not 1.
6. `LHV` = number of units of storage occupied by the hash-table (see Section 6.2.3 [Hash-Tables and their Definition], page 43).

Special action is necessary if any of these constants are not integers (one solution may be to 'devalue' the storage unit). The length-defining macros only occur within the argument of the `OF` macro, but it will usually be convenient to give them global scope.

To illustrate sample values of these constants, in the PDP-7 implementation the first five all had value one, whereas in the System/360 implementation `LPT` and `LNM` had value four but `LSW` and `LCH` had value one.

The specification of the `OF` macro is as follows:

*Purpose*

Designates numerical constants that are dependent on the amount of storage occupied by individual data types.

*General Form*

```
OF ( argument )
```

*Structure Representation*

```
OF WITHS ( )
```

*Examples*

a. `OF(LPT)`
b. `OF(2*LPT - LSW)`

*Restrictions*

The argument is such that if all the length-defining macros are replaced by their numerical values, then what results is a macro expression involving only constants (this macro expression will in practice have a positive result).

*Action*

`OF` should be replaced by the result of the macro expression occurring as its argument. For most L-maps the following macro should suffice:

```
MCDEF OF WITHS ( ) AS <%%A1..>
```

### 3.3.2 The Quote macro

*Purpose*

Designates a character constant.

*General Form*

`'character(s)'`

*Structure Representation*

`' '`

(if it is a macro, it should be a straight-scan macro, and the argument should be referenced as `%WB1.`).

*Examples*

1. `'A'`
2. `'$'`
3. `' '`
4. `'MCDEF'`

*Restrictions*

Where quote is used in the program SECTIONs its argument is always a single character. However it may have any atom as an argument if it occurs in the data SECTIONs. The characters that can occur within the argument are the upper case letters A to Z, a space, or any of the following:

```
, = ( ) + - / */ ; $
```

Each character stands for itself except `$`, which stands for newline.

*Notes*

a. If the object language contains some facility for literal character constants then the quote macro should not require much work during an L-map. Only `$` will need replacing and this can be achieved by:

```
MCDEF <' WITH $ WITH '> AS < internal code for newline>
```

at least within the program SECTIONs.

b. The character set of an implementation is arbitrary except that it must contain all the characters that can occur within the argument to the quote macro (or their equivalents), together with the digits 0 to 9. Furthermore the character set must not contain 'shift' characters, and the internal representation of characters may need to be changed if such characters normally exist.

### 3.3.3 The `AD` and `BLOCK` macros

The `AD` and `BLOCK` macros are used to represent constants of type pointer. Since the `BLOCK` macro is only used within block moving statements, its description is delayed until these statements are described in Section 4.3 [Block Moving Statements], page 25. The `AD` macro is described below.

*Purpose*

Designates address of data label.

*General Form*

```
AD( identifier )PT
```

*Structure Representation*

```
AD WITHS ( ) WITHS PT
```

*Example*

```
    AD(KSPACE)PT
```

*Restrictions*

The identifier is the name of a data label.

*Action*

`AD` should be replaced by a literal representing the value of the address.

*Notes*

In some L-maps `AD` may prove very difficult to encode. It may help to build it into the expression evaluation macro (see Section 3.6 [Arithmetic Expressions], page 16).

### 3.3.4 Constants Represented by Identifiers

Some constants in L are represented by identifiers. This is done either for mnemonic purposes (e.g. the constant `TRUE`) or because the value of a constant will vary between different machines (e.g. the constant `STOPCODE`). A complete list of such constants follows, together with a description of what values should be used to replace them.

#### 3.3.4.1 Switch Constants

There are two possible switch constants:

| | | |
|---|---|---|
| a. | `TRUE.` | This has value one. |
| b. | `FALSE.` | This has value zero. |

#### 3.3.4.2 Pointer Constants

There are two possible pointer constants:

| | | |
|---|---|---|
| a. | `ZEROPT.` | This may have any value which is less than or equal to any possible pointer value. |
| b. | `NULLPT.` | This may have any value that does not correspond to a possible value of a pointer. |

In most L-maps both `ZEROPT` and `NULLPT` will have value zero.

#### 3.3.4.3 Character Constants

There is one possible character constant:

|  |  |  |
|---|---|---|
| a. | `STOPCODE.` | This is a special marker, not representing any possible character, which uniquely identifies the end of the source text. `STOPCODE` is used by the `READ` statement, see Section 4.4.1 [The `READ` Statement], page 28. |

#### 3.3.4.4 Number Constants

The following constants represent the sizes of the blocks of variables declared using the `BLOCKDEC` statement (see Chapter 5 [Declarative Statements and Initial Values], page 37).

|  |  |  |
|---|---|---|
| a. | `SDBSZ.` | This represents the number of storage units used by the `SDB` block. |
| b. | `OPDBSZ.` | This represents the number of storage units used by the `OPDB` block. |
| a. | `ALLSZ.` | This represents the number of storage units used by the `ALL` block. |
| a. | `EDBSZ.` | This represents the number of storage units used by the `EDB` block. |

The following mnemonic markers are used to identify the various types of operation macro and insert:

|  |  |  |
|---|---|---|
| e. | `OPMK.` | This identifies an ordinary operation macro and has value 0. |
| f. | `LOCMK.` | This identifies a local NEC macro and has value 1. |
| g. | `UINSMK.` | This identifies an unprotected insert and has value 2. |
| h. | `PINSMK.` | This identifies a protected insert and has value 3. |
| i. | `STRMK.` | This identifies a straight-scan macro and has value 4. |

The following markers may follow the specification of a delimiter (in the case of this set of markers, as for the previous set, the implementor need not, in fact, be concerned with their meanings; he only needs to know what to replace them by).

|  |  |  |
|---|---|---|
| j. | `ENDCHN.` | This identifies a closing delimiter and has value zero. It also serves as an end-of-chain marker. |
| k. | `EXCLMK.` | This identifies an exclusive delimiter. |
| l. | `WITHMK.` | This is used in implementing the `WITH` keyword in ML/I. |
| m. | `WTHSMK.` | This is used in implementing the `WITHS` keyword in ML/I. |
| n. | `SPCSMK.` | This is used in implementing the `SPACES` keyword in ML/I. |

Four distinct values should be chosen for `EXCLMK`, `WITHMK`, `WTHSMK` and `SPCSMK`. Each value should be a number larger in absolute value than the largest possible workspace size (see Section 2.8 [Use of Storage and Stacks], page 8).

Finally there are two constants dependent on the value N described in Chapter 6 of the *ML/I User's Manual*. The value of N is dependent on the width of listings but a typical value would be 30.

|  |  |  |
|---|---|---|
| o. | `TEXMAX.` | This has value 2N. |
| p. | `HTMAX.` | This has value N-4. |

## 3.4 Indirect Addresses

In addition to constants and variables, statements in L may involve indirect addresses. These are represented by the `IND` macro, which is described below.

*Purpose*

Specifies indirect address.

*General Form*

```
IND ( arithmetic expression ) (CH)
                              (NM)
                              (PT)
                              (SW)
```

(Arithmetic expressions are defined in Section 3.6 [Arithmetic Expressions], page 16).

*Structure Representation*

```
IND WITH ( OPT ) WITH CH OR ) WITH NM OR ) WITH PT OR ) WITH SW ALL
```

*Examples*

a. `IND(SPT)CH`

b. `IND(HASHPT + TEMP - OF(LCH))SW`

c. `IND(AD(KSPACE)PT)NM`

*Restrictions*

The arithmetic expression must not itself contain an indirect address. The result of the arithmetic expression is a pointer.

*Action*

`IND` specifies an item of the data type given by the last two letters, which is accessed indirectly via the pointer. This item will lie in workspace or in the data SECTIONs.

*Notes*

The mapping macro for `IND` normally requires to be carefully planned. It should only be called from within another macro and this macro should pass down to the `IND` macro an indication as to what to do with the indirectly addressed item (e.g. add it, store into it, etc.). It is often convenient to build `IND` into the expression evaluation macro, which will be outlined later.

## 3.5 Notation for Describing Arguments

It is necessary at this stage to define a notation that will be needed in subsequent Sections to specify the form and type of arguments. In this notation an argument will be specified by writing:

*`form-type`*

where *form* consists of a sequence of one or more of the following letters:

| | |
|---|---|
| `V` | for variable |
| `C` | for constant |
| `I` | for indirect address |

and *type* consists of one or more of the letter pairs `CH`, `NM`, `PT`, `SW`. These letter pairs represent the data types. *form* and *type* indicate the various forms and types an argument may take. For example `VI-SW` means a variable or indirect address of type switch and `C-NMPT` means a constant of type number or pointer.

## 3.6 Arithmetic Expressions

Several statements in L allow arithmetic expressions as arguments. There is no explicit expression evaluation macro, but an artificial macro for this purpose will need to be set up as outlined below. The specifications of arithmetic expressions are as follows:

*General Forms*

a.
```
<<I-NMPT ?>> <<(+) V-NMPT *?>> <<(+) C-NM >>
              (-)               (-)
```
where at least one constituent is not null.

b. `C-PT`

*Examples*

a. `-6`

b. `IND(SPT)NM + IDLEN - 3`

c. `- SKVAL + OF (LPT+LSW)`

d. `SPT - ARGNO - SDBSZ`

e. `AD (KSPACE)PT`

*Action*

Evaluate the arithmetic expression by the normal rules of arithmetic. The result of the expression will be a number or a pointer (adding or subtracting a number or a pointer yields a pointer, and subtracting two pointers yields a number).

*Notes*

Calls of the macro for expression evaluation must be artificially set up by other macro calls. Thus for example the `SET` macro might call:

```
EXPR %AT1.
```

where `EXPR` was the expression evaluation macro. In this case `EXPR` might have the structure representation:

```
OPT EXPR OR EXPR WITHS - ALL N1 OPT + N1 OR - N1 OR NL ALL
```

(`EXPR` would need to be called on a subsequent pass to the `SET` macro, or the technique for creating dynamic macro calls described in Chapter 7 of the *ML/I User's Manual* would need to be used.) It might be found desirable to build the `IND` and `AD` macros into the `EXPR` macro by making the `EXPR` macro recognise occurrences of these within its arguments.

# 4 Executable Statements

This chapter describes the statements that occur in the program SECTIONs. These are divided into five categories as follows:

a. Statements for communicating with routines.
b. Compound statements.
c. Block moving statements.
d. I/O statements.
e. Assignment and branching statements.

## 4.1 Routines

L contains two types of routine, namely *subroutines* and *linkroutines*. These are declared using the `SUBROUTINE` and `LINKROUTINE` statements and are both called using the `CALL` statement. Declarations of subroutines or linkroutines are always global and are hence never nested within other declarations. Declaration need not precede use, although it would be possible to rearrange the logic of ML/I to make this so.

The following Sections describe subroutines and linkroutines together with how they are declared and how they return when their actions are completed. After this the `CALL` statement will be described.

### 4.1.1 Subroutines

Subroutines in L either have a single parameter, which may be of type number, switch or pointer, or no parameter at all. In addition some subroutines require an *exit label* to be supplied with each call. An exit label specifies a label to which the subroutine is to go if some special condition arises.

The calls of subroutines always match their declarations in that if a subroutine has a parameter it is always called with an argument of the same type and if it is declared to have an exit label it is always called with one. A subroutine never changes the value of its parameter.

A return from a subroutine is accomplished by the `RETURN FROM` or `EXIT FROM` statements. These two statements, together with the `SUBROUTINE` statement, are described below.

#### 4.1.1.1 The `SUBROUTINE` statement

*Purpose*

Declares a subroutine.

*General Form*

```
SUBROUTINE identifier << (PARPT) ?>> <<EXIT subsidiary comment ?>>
                         (PARNM)
                         (PARSW)

<< statement *>>

ENDSUB
```

*Structure Representation*

Preferably two separate macros.

1. `SUBROUTINE OPT ( ) N1 OR N1 EXIT N2 OR N2 NL ALL`
2. `ENDSUB NL`

*Examples*

The following are examples of first lines of subroutine declarations:

a. `SUBROUTINE CMPARE (PARPT) EXIT //COMPARISON FAILS//`
b. `SUBROUTINE NOARGS`

*Restrictions*

Subroutines are never called recursively.

*Action*

Set `%A1.` as the name of a subroutine. At the start of the subroutine generate code to assign the argument (if any) to the parameter and, if necessary, preserve the exit label and the return address in order that they may be available to the `RETURN FROM` and `EXIT FROM` statements.

*Notes*

a. The last two letters of the parameter indicate its type. `PARPT`, `PARSW` and `PARNM` are declared in the `VARS` SECTION of the logic just like other variables. They could in fact be equated to one another in an L-map where types were not differentiated, as the logic of ML/I is such that there is never more than one parameter in existence at any one time, i.e. if a subroutine `A` with a parameter calls a subroutine `B` with a parameter then the logic is such that it does not matter if the parameter of `A` is clobbered as a result of the call of `B`.
b. There are basically two possible techniques for preserving return addresses. Either each subroutine can have a unique storage location for preserving its return address or a stack of return addresses (which must be entirely separate from the other stacks) can be used. A problem with the latter approach is that some `GOTO` statements jump out of subroutines into the main logic without passing through the normal exit mechanism. To help solve this problem each label referenced by such a `GOTO` is preceded by a statement prefix `CSS`, which can be mapped into instructions to clear the subroutine stack.
c. The comment following `EXIT` is merely an aid to the reader of the logic.

#### 4.1.1.2 The `RETURN FROM` statement

*Purpose*

Return from a subroutine.

*General Form*

`RETURN FROM` ***`identifier`***

*Structure Representation*

```
RETURN WITHS FROM NL
```

*Example*

```
RETURN FROM CMPARE
```

*Restrictions*

`RETURN FROM` statements always lie within a subroutine, the name of which is given by `%A1.`

*Notes*

Return from the subroutine to the point of call.

#### 4.1.1.3 The `EXIT FROM` statement

*Purpose*

Returns from a subroutine to an exit label.

*General Form*

`EXIT FROM` ***`identifier`***

*Structure Representation*

```
EXIT WITHS FROM NL
```

*Example*

```
EXIT FROM CMPARE
```

*Restrictions*

`EXIT FROM` statements always lie within a subroutine, the name of which is given by `%A1.`, and this subroutine has an exit label.

*Action*

Go to the program label supplied as an exit label in the call of the subroutine.

*Notes*

Exit labels may be implemented in many different ways. Two possibilities are:

1. To treat the exit label as an extra argument of the subroutine.
2. (when mapping into assembly language)
   to code a statement of form:

   `CALL` *`SUB`* `EXIT` *`LAB`*

   as

```
CALL SUB
GO TO LAB
```

In this case the `EXIT FROM` statement should cause a return to the point of call and the `RETURN FROM` statement, within a subroutine with an exit label, should cause a return to the point of call plus a suitable offset to skip over the `GO TO` statement.

### 4.1.2 The linkroutine

The logic contains only one linkroutine, which is named `STKARG`. The mechanism of linkroutines is included in L to cater for recursion, and `STKARG` may be regarded as a recursive subroutine. It has no parameter or exit label and is called by the statement:

```
CALL STKARG
```

The difference between linkroutines and subroutines is that in subroutines the preserving of the return address, if necessary, is the responsibility of an L-map whereas in a linkroutine the return address must be assigned to the variable `LINKPT` when the routine is called. `LINKPT` is one of the variables automatically preserved by the logic in recursive situations, and hence the implementor does not need to bother about the problem. A return from `STKARG` is accomplished by the `LINK BACK` statement.

If the object language is an assembly language then `STKARG` will normally be represented as a subroutine and will present few problems. However in L-maps into high-level languages it will usually not prove possible to represent `STKARG` as a subroutine because:

a. Return addresses are often not directly accessible in high-level languages,
b. `STKARG`, unlike a subroutine, contains a jump to a point outside itself, from where a jump back into `STKARG` subsequently occurs (see Section 4.5.3 [The `GO TO` Statement], page 32).

To surmount these difficulties it may be necessary to represent `STKARG` as a piece of labelled code which is ‘called’ by setting `LINKPT` as the return address (or as an index to a **switch**, in the ALGOL sense, of possible return addresses) and then performing the statement:

```
GO TO STKARG
```

Moreover it may be necessary to treat `LINKPT` as a special type of pointer since it points at the program rather than the data.

#### 4.1.2.1 The `LINKROUTINE` statement

*Purpose*

Declares a linkroutine.

```
LINKROUTINE identifier

<< statement >>

ENDSUB
```

*Structure Representation*

Assuming `ENDSUB` to be a separate macro, the structure representation of the `LINKROUTINE` statement is

```
    LINKROUTINE NL
```

*Example*

```
LINKROUTINE STKARG
. . .
ENDSUB
```

*Action*

Set the identifier as a subroutine entry point or a label, as appropriate, and generate code to store the return address in `LINKPT`.

*Notes*

It is possible, by defining `CALL STKARG` as a macro, to set `LINKPT` at the point of call instead of when the linkroutine is entered.

#### 4.1.2.2 The `LINK BACK` statement

*Purpose*

Returns from a linkroutine.

*General Form*

```
LINK BACK
```

*Structure Representation*

```
LINK WITHS BACK NL
```

*Restrictions*

`LINK BACK` occurs only within the linkroutine `STKARG`.

*Action*

Return to the address contained in `LINKPT`.

### 4.1.3 The `CALL` statement

*Purpose*

Calls a routine.

*General Forms*

```
 a. CALL identifier << EXIT program label ?>>
 b. CALL identifier (arithmetic expression)(NM)
                                           (PT)
          << EXIT program label ?>>
 c. CALL identifier (VIC-SW)SW << EXIT program label ?>>
```

*Structure Representation*

```
CALL OPT ( ) N1 OR N1 EXIT N2 OR N2 NL ALL
```

*Examples*

a. `CALL CMPARE ( IDPT + 1 )PT EXIT NOTFND`

b. `CALL NOARGS`

*Restrictions*

The call must correspond to some `SUBROUTINE` or `LINKROUTINE` declaration in the MI-logic, or to a subroutine in the MD-logic.

*Action*

Call the routine. If it has an argument, load the value (not the address) of the argument into some fixed place (e.g. an accumulator) in order that it can be assigned to the corresponding parameter. If there is an exit label, make this available to the called routine.

*Notes*

In some L-maps it may be convenient to assign the argument to the corresponding parameter at the point of call, rather than to load it into an accumulator and then store this accumulator into the parameter when the routine is entered.

## 4.2 Compound Statements

There are two compound statements in L, namely `IF` and `CHAIN FROM`. These are described below.

### 4.2.1 The `IF` statement

*Purpose*

Conditional statement.

*General Form*

a. `IF` ***`condition`*** `THEN` ***`statement`***

b.
```
IF condition THEN
<< statement *>>
END
```

*Structure Representation*

See later.

*Examples*

a. `IF SPT = IDPT THEN GO TO ENDID`

b.
```
IF IND(SPT)CH NE 'A' THEN
. . .
END
```

*Restrictions*

See next Section for the form of a *condition*.
In form a) the *statement* is never an `IF` or `CHAIN FROM` statement. Statements of form b) are never nested within one another.

*Action*

Perform the statement(s) if the condition holds.

*Notes*

a. It is probably best to consider `END` as a separate macro from `IF`.

b. In form a) the closing newline acts as a closing delimiter of both the `IF` statement and the statement following `THEN`. There are three possible ways of dealing with this:

   1. Turn form a) into form b) (or something of similar syntax) on a prepass.
   2. Treat newline as an exclusive delimiter of all statements except `IF` and `CHAIN FROM`.
   3. Make `IF` a straight-scan macro and, when dealing with form a), insert the last argument in the following way:

      ```
      MCDEF ZZ AS <%WAT1.
      >
      ZZ
      ```

   The first approach is probably the simplest.

c. Many L-maps will wish to recognise `THEN GO TO` as a special case, in order to generate better code.

### 4.2.1.1 The Form of an `IF` Condition

*General Form*

The condition of an `IF` statement can have the following form:

```
a. relation
b. relation << & relation *>>
c. relation << | relation *>>
```

where `&` means 'and' and `|` means 'or' (note that 'and' and 'or' cannot both occur in the same condition).
A *relation* can have the following forms:

```
a. arithmetic expression (GR) VCI-PTNM
                         (GE)
                         (LE)
                         (= )
                         (NE)

b. VI-SW (= ) VIC-SW
         (NE)
```

c. `I-CH (= ) IC-CH`
   `(NE)`

`GR` means 'greater than', `GE` means 'greater than or equal to', `LE` means 'less than or equal to', and `NE` means 'not equal to'.

*Examples*

a. `IF SPT - IDPT GE 6 | IND(PARPT)SW NE 3 THEN ...`

b. `IF AAA = 1 & BSW NE 2 & CCC + DDD GR EEE THEN ...`

*Restrictions*

In form a) of a relation the arithmetic expression never contains a constant as a constituent, i.e. constants only appear after rather than before a relational operator. If an indirect address appears after the relational operator, this always takes the form `IND(V-PT)...`

*Action*

Test whether the condition holds.

*Notes*

The data types to be compared can be found by examining the last two characters of the first argument.

#### 4.2.1.2 Structure Representation for the `IF` Statement

The structure representation corresponding to the `IF` statement will vary considerably between L-maps. If `THEN GO TO` was to be recognised as a special case then a prepass might perform the transformation:

```
MCDEF THEN WITHS GO WITHS TO AS THENGO
MCSKIP D,THEN WITHS NL
MCDEF THEN NL AS <<THEN>
%A1.
END
>
```

In this case the delimiter structure corresponding to the `IF` and `END` statements could consequently be:

1. `IF N1 OPT GR OR GE OR = OR NE OR LE ALL OPT THEN WITH NL OR THENGO NL`
   `OR | N1 OR & N1 ALL`
2. `END NL`

### 4.2.2 The `CHAIN FROM` statement

*Purpose*

Follows down a chain where the next link is given by its offset from the current link.

*General Form*

```
        CHAIN FROM arithmetic expression EXIT program label
        << statement *?>>
        ENDCH
```

`ENDCH` is regarded as a separate statement in that it may be preceded by the placing of a program label.

*Structure Representation*

Two macros:

1. `CHAIN WITHS FROM EXIT NL`
2. `ENDCH NL`

*Example*

```
CHAIN FROM IDPT + OF(LNM) EXIT JC1
   SET IND(CHANPT)NM = 0
ENDCH
```

*Restrictions*

The arithmetic expression yields a pointer as value. Calls of the `CHAIN FROM` statement are never nested within one another.

*Action*

Taking `CHAIN FROM` as two macros as indicated in its suggested structure representation, the first macro is equivalent to:

```
             SET CHANPT = %A1.
             IF CHANPT = NULLPT THEN GO TO %A2.
      [Lab]  SET CHLINK = IND(CHANPT)NM
```

and the `ENDCH` macro is equivalent to:

```
      IF CHLINK NE ENDCHN THEN
         SET CHANPT = CHANPT + CHLINK
         GO TO Lab
      END
```

where `Lab` is some generated label unique to the call of `CHAIN FROM`, and `ENDCHN` and `NULLPT` are the constant-defining macros described in Section 3.3.4 [Constants Represented by Identifiers], page 13.

## 4.3 Block Moving Statements

L contains three statements for moving about contiguous blocks of information. These statements are `MOVE FROM`, `MSTACK FROM` and `MUNSTACK FROM`. Each statement requires three arguments as follows:

a. A pointer to the start of the block to be moved.
b. A pointer to the start of the area to be moved to.
c. A length of the block to be moved.

(in the case of `MUNSTACK FROM`, argument b) is implicit). Argument c) is always represented by an arithmetic expression, yielding a positive (non-zero) numerical value, and each of arguments a) and b) may be represented in either of two ways, depending on the area pointed at:

a. If the area pointed at is in workspace or the data SECTIONs then the argument is specified by an arithmetic expression yielding a pointer as result.
b. If the area pointed at is in the `VARS` SECTION then the argument is specified by:

   ```
   BLOCK ( name )
   ```

   where the *name* is the name of a block of variables declared by the `BLOCKDEC` statement (see Chapter 5 [Declarative Statements and Initial Values], page 37). `BLOCK` can be regarded as a constant-defined macro of type pointer. It is only used within arguments to block moving statements. Many L-maps will treat `BLOCK` in exactly the same way as the `AD` macro (see Section 3.3.3 [The `AD` and `BLOCK` macros], page 12).

The descriptions of the three block moving statements are written in terms of loops using the local variables `LV1`, `LV2`, `LV3` and `LV4`. However an L-map need not follow the exact descriptions of these statements, provided that the overall effect is not altered. In particular the variables `LV1`, `LV2`, `LV3` and `LV4` need not be used. In fact the whole purpose of the block moving statements is to allow efficient code specially tailored to the object machine to be inserted.

### 4.3.1 The `MOVE FROM` Statement

*Purpose*

Moves a block of information from one place to another.

*General Form*

```
MOVE FROM arg A TO arg B LENG arg C << BACKWARDS ?>>
```

*Structure Representation*

```
MOVE WITHS FROM TO LENG OPT BACKWARDS N1 OR N1 NL ALL
```

*Example*

```
MOVE FROM BLOCK(SDB) TO IDPT + OF(LNM) LENG SDBSZ
```

*Restrictions*

See general description of block moving statements.

*Action*

a. In the `BACKWARDS` case, which is used only when two fields may overlap and upset a forwards move, the action is equivalent to:

```
            SET LV1 = %A1.
            SET LV2 = LV1 + %A3. - OF(LCH)
            SET LV3 = %A2. + %A3. - OF(LCH)
      [XX]  SET IND(LV3)CH = IND(LV2)CH
            IF LV2 NE LV1 THEN
               SET LV2 = LV2 - OF(LCH)
               SET LV3 = LV3 - OF(LCH)
               GO TO XX
            END
```

b. If the `BACKWARDS` option is not specified, a forwards move is performed in the following way:

```
             SET LV2 = %A1.
             SET LV3 = %A2.
             SET LV1 = LV2 + %A3. - OF(LCH)
      [YY]   SET IND(LV3)CH = IND(LV2)CH
             IF LV2 NE LV1 THEN
                SET LV2 = LV2 + OF(LCH)
                SET LV3 = LV3 + OF(LCH)
                GO TO YY
             END
```

### 4.3.2 The `MSTACK FROM` Statement

*Purpose*

Stacks a block of information.

*General Form*

```
MSTACK FROM arg A LENG arg C ON (FSTACK)
                                (BSTACK)
```

*Structure Representation*

```
MSTACK WITHS FROM LENG ON NL
```

*Example*

```
MSTACK FROM IDPT LENG IDLEN ON FSTACK
```

*Restrictions*

See general description of block moving statements.

*Action*

a. If `%A3.` is `BSTACK`, the action is equivalent to the statements:

```
CALL DECLF ( %A2.)NM
MOVE FROM %A1. TO LFPT LENG %A2.
```

b. If `%A3.` is `FSTACK`, the action is equivalent to the statements:

```
SET LV4 = FFPT
CALL BUMPFF (%A2.)NM
MOVE FROM %A1. TO LV4 LENG %A2.
```

### 4.3.3 The `MUNSTACK FROM` Statement

*Purpose*

Unstacks a block of information from the backwards stack.

*General Form*

```
MUNSTACK FROM arg A TO arg B LENG arg C FROM BSTACK
```

*Structure Representation*

```
MUNSTACK WITHS FROM TO LENG FROM WITHS BSTACK NL
```

*Example*

```
MUNSTACK FROM STAKPT TO BLOCK(EDB) LENG 3
```

*Restrictions*

See general description of block moving statements.

*Action*

The action is equivalent to the statements:

```
MOVE FROM %A1. TO %A2. LENG %A3.
SET LFPT = %A1. + %A3.
```

(note that `%A1.` might be `LFPT` and hence the ordering of the two above statements should not be changed to try and improve efficiency.)

## 4.4 I/O Statements

There are three statements in L that deal with I/O, namely `READ`, `OUTPUTID` and `PRTEXT`. These statements deal respectively with the input text, the output text and error messages. In addition certain of the machine-dependent subroutines deal with the production of error messages (see Section 7.2 [Subroutines for Error Messages], page 48). `READ` and `OUTPUTID` each occur only once in the logic. However they have been represented as statements in L rather than as machine-dependent subroutines in order that, for reasons of efficiency, they can be replaced by in-line code on an L-map (if the object language has a macro facility, it may be convenient, for ease of changing, to map I/O statements into object language macros).

ML/I has a single stream of input. However there may be several streams of output as follows:

a. Output text in a form that can be read back in.
b. Listing of output text.
c. Listing of input text.
d. Listing of error messages and other diagnostic information (this is called the *debugging file* in the *ML/I User's Manual*).

Alternatives b) and c) are optional and need not be available on all implementations of ML/I.

### 4.4.1 The `READ` Statement

*Purpose*

Reads in the input text.

*General Form*

```
READ
```

*Structure Representation*

```
READ NL
```

*Action*

Read one character of input and translate it to internal code. If appropriate check that the character is legal and, if not, perform the action:

```
CALL ERSIC
```

and replace the character by the *error character* for the implementation. Then place the character on the top of the forwards stack by performing the following action:

```
SET IND(FFPT)CH = character

//UPDATE STACK POINTER//

SET FFPT = FFPT + OF(LCH)

//TEST FOR OVERFLOW//

IF FFPT GE LFPT THEN GO TO ERLSO
```

The `READ` statement should make each line of input end with the character 'newline'. In the case of card input this character may need to be added to each line and in the case of paper tape input the pair of characters 'carriage return, line feed' may need to be mapped into the single character 'newline'.

If appropriate the `READ` statement should provide a listing of the input with accompanying line numbers. When the input is exhausted the `READ` statement should return the character represented by `STOPCODE` (see Section 3.3.4.3 [Character Constants], page 14). The logic is arranged so that if a `STOPCODE` is returned then the `READ` statement is not executed again but control goes to the label `MDHALT` (see Section 7.3.3 [The `MDHALT` Label], page 51) instead. If the input is split up into several physical units (e.g. several separate paper tapes), the `READ` statement should take care of the interface between different units.

The `READ` routine also has responsibility for taking action at the boundary between lines. At the start of each line, including the first, the following action should be performed:

```
SET S2 = S2 + 1
IF LINECT = TLINCT THEN SET TLINCT = S2
SET LINECT = S2
IF S2 = 1 THEN insert startline character
               at beginning of line
```

where `S1` and `S2` are S-variables. This action should *not* be performed on the very last line, i.e. between the last newline and the `STOPCODE` at the end.

Any unused internal code can be chosen for startline. This is defined by the `LAYCHAIN` statement (see Section 6.2.2 [The `LAYCHAIN` statement], page 42).

To summarise, the action of the `READ` statement should be to make the input text appear as a single sequence of characters, with the character 'newline' at the end of each line, the character `STOPCODE` at the end of the text, and startline characters inserted where appropriate.

### 4.4.2 The `OUTPUTID` Statement

*Purpose*

Produces output.

*General Form*

```
OUTPUTID
```

*Structure Representation*

```
OUTPUTID NL
```

*Action*

Output a sequence of characters. The sequence is pointed at by `IDPT` and the number of characters in the sequence is given by `IDLEN`. `IDLEN` is always greater than zero and can be arbitrarily large. The values of `IDPT` and `IDLEN` may be clobbered by the `OUTPUTID` statement.

Normally output should be sent to a device that can then be read back in by the job step that comes after macro processing. It may also be desirable to produce a listing of the output, on option.

`OUTPUTID` should, if necessary, translate characters to the code required by the external device. It may be necessary to take special action with the character 'newline'. The marker `STOPCODE` is never sent to the `OUTPUTID` routine. Instead it is the responsibility of the code at `MDHALT` (see Section 7.3.3 [The `MDHALT` Label], page 51) to finalise the output. Startline characters should be ignored.

### 4.4.3 The `PRTEXT` Statement

*Purpose*

Prints literal strings within error messages.

*General Form*

```
PRTEXT [ << character *>> ]
```

*Structure Representation*

```
PRTEXT WITHS [ N1 OPT $ N1 OR ] WITH NL ALL
```

*Example*

```
PRTEXT[$PROCESS ABORTED$]
```

*Restrictions*

The set of possible characters occurring within the argument to `PRTEXT` is the same as for the quote macro (see Section 3.3.2 [The Quote macro], page 12).

*Action*

Print the argument as a string of characters on the error message listing. Each character stands for itself except `$`, which represents a newline.

*Notes*

a. `PRTEXT` should be mapped with a straight-scan macro. Arguments should be inserted using `%WB1.` etc. It is usually convenient to deal with `$` by treating it as a delimiter of `PRTEXT` as suggested by the above structure representation.

## 4.5 Assignment and Branching Statements

A list of all the L statements used to perform assignment or branching operations follow. The statements are arranged in alphabetical order.

### 4.5.1 The `BACKSPACE` Statement

*Purpose*

Examines former value of a variable now lying on `BSTACK`.

a. `BACKSPACE` *`V-NMPTSW`*
b. `BACKSPACE` *`V-NM`* `GIVING` *`V-NM`*
c. `BACKSPACE` *`V-PT`* `GIVING` *`V-PT`*
d. `BACKSPACE` *`V-SW`* `GIVING` *`V-SW`*

*Structure Representation*

```
BACKSPACE OPT GIVING N1 OR N1 NL ALL
```

*Examples*

a. `BACKSPACE SPT`
b. `BACKSPACE SPT GIVING IDPT`

*Restrictions*

`%A1.` is in the block that is called `SDB` (see `BLOCKDEC` statement of Chapter 5 [Declarative Statements and Initial Values], page 37).

*Action*

Let `N` be the offset of `%A1.` from the start of the `SDB` block (i.e. the number of units of storage occupied by the variables preceding `%A1.` in the `SDB` block). Then the action taken for the respective forms is:

a. `SET TEMPT = DBUGPT + N`
b. `SET %A2. = IND(DBUGPT + N)NM`
c. `SET %A2. = IND(DBUGPT + N)PT`
d. `SETSW %A2. = IND(DBUGPT + N)SW`

In cases b) to d) `TEMPT` may be clobbered.

### 4.5.2 The `CHARMATCH` Statement

*Purpose*

Multi-way conditional branch statement.

*General Form*

`CHARMATCH` *`V-PT`* `<<,` *`C-CH`* `GOING` *`program-label`* `*>>`

*Structure Representation*

```
CHARMATCH N1 OPT , GOING N1 OR NL ALL
```

*Example*

```
CHARMATCH IDPT, 'A' GOING INSA, 'B' GOING INSB
```

*Action*

The action is equivalent to:

```
IF IND(%A1.)CH = %A2. THEN GOTO %A3.
IF IND(%A1.)CH = %A4. THEN GOTO %A5.
. . .
IF IND(%A1.)CH = %AT1-1. THEN GOTO %AT1.
```

It is quite possible for none of the above tests to hold.

### 4.5.3 The `GO TO` Statement

*Purpose*

Branch statement.

*General Form*

`GO TO` *`identifier`*

*Structure Representation*

```
GO WITHS TO NL
```

*Example*

```
GO TO BSNEXT
```

*Restrictions*

The identifier is a program label or a label in the MD-logic. A `GO TO` statement never goes into a subroutine from outside (it may, however, go out of a subroutine to a label outside and it may go into the linkroutine from outside and vice versa).

*Action*

Branch to designated label.

### 4.5.4 The `SCALE` Statement

*Purpose*

Multiplies a variable by a constant.

*General Forms*

a. `SCALE` *`V-NM`* `BY` *`C-NM`*

b. `SCALE` *`V-NM`* `BY` *`C-NM`* `GIVING` *`V-NM`*

*Structure Representation*

```
SCALE BY OPT GIVING N1 OR N1 NL ALL
```

*Example*

```
SCALE MEVAL BY OF(LNM) GIVING ARGNO
```

*Restrictions*

`%A2.` is a small positive constant (not greater than `2*LPT`).

*Action*

Multiply `%A1.` by `%A2.`, and assign the result to `%A3.` if `%A3.` exists; otherwise to `%A1.`

*Notes*

The constant will always be a call of the `OF` macro (see Section 3.3.1 [The `OF` and length-defining macros], page 11) and it may often have value 1, in which case the replacement text for form a) could be null.

### 4.5.5 The `SET` Statement

*Purpose*

Assigns a value to a number or a pointer.

*General Form*

a. `SET` *`VI-NMPT`* `<<,` *`V-NMPT`* `*?>> =` *`arithmetic expression`*

b. `SET` *`VI-NM`* `=` *`VI-SW`*

*Structure Representation*

```
SET N1 OPT , N1 OR = ALL NL
```

*Examples*

a. `SET IND(SPT)NM, ARGLEN = IND(IDPT)NM+TEMP-6`

b. `SET SKVAL = - SKVAL + 1`

c. `SET TYPE = INSW`

d. `SET IDLEN = IDLEN + IDPT - SPT`

*Restrictions*

All the quantities to the left of the equals sign are of the same type. If any quantity occurs on both the left and the right of the equals sign then it must be the first quantity on the right, as illustrated by examples c) and d) above.

*Action*

Evaluate the arithmetic expression and assign its value to the variables and/or indirect address on the left of the equals sign.

### 4.5.6 The `SETSW` Statement

*Purpose*

Assigns a value to a switch.

*General Forms*

```
a. SETSW VI-SW <<, V-SW *?>> = VIC-SW
b. SETSW VI-SW = VI-SW (&) VC-SW
                       (|)
```

*Structure Representation*

```
SETSW N1 OPT , N1 OR = ALL OPT | N2 OR & N2 OR N2 NL ALL
```

*Examples*

```
SETSW IND(TEMPT)SW, PARSW = 1
SETSW COPDSW = IND(INFOPT)SW & 1
```

*Restrictions*

If form b), `%A1.` cannot be the same as `%A3.` (e.g. the form:

```
ASW = arg & ASW
```

cannot occur).

*Action*

Evaluate the expression to the right of the equals sign (`|` means 'inclusive or' and `&` means 'and') and assign its value to the variable(s) and/or indirect address on the left.

### 4.5.7 The `STACK` Statement

*Purpose*

Stacks individual values.

*General Form*

```
STACK << value (type) *>> ON (FSTACK)
                             (BSTACK)
```

where *value* is one of the following forms:

a. `VCI-SW` in which case *type* is `SW`.
b. *arithmetic expression* in which case *type* is `NM` or `PT`.

*Structure Representation*

```
STACK N1 OPT ( ) N1 OR ON ALL NL
```

*Example*

```
STACK PARSW(SW) 3(NM) TRUE(SW) IDPT-1(PT) ON FSTACK
```

*Restrictions*

In the `FSTACK` case, if the forwards stack pointer (`FFPT`) occurs within a *value* it occurs within the first value. In the `BSTACK` case the backwards stack pointer

(`LFPT`) is never used (these two restrictions make optimisation of multiple stacking operations possible, though care must be taken not to bump `FFPT` until after the first value has been calculated).

*Action*

a. If the last argument is `FSTACK`, the following operation should be performed for each *value* and *type*:

   1. If *type* is `NM` or `PT` then

      ```
      SET IND(FFPT)type = value
      ```

   2. If *type* is `SW` then

      ```
      SETSW IND(FFPT)SW = value
      ```

   followed in each case by

   ```
   CALL BUMPFF (N )NM
   ```

   where `N` is the value of `OF(L type)` — see Section 3.3.1 [The `OF` and length-defining macros], page 11). Thus, for example:

   ```
   STACK IDPT - 1(PT) ON FSTACK
   ```

   is equivalent to:

   ```
   SET IND(FFPT)PT = IDPT - 1
   CALL BUMPFF(OF(LPT))NM
   ```

b. In the `BSTACK` case, the following operations should be performed for each *value* and *type*:

   ```
   CALL DECLF ( N )NM
   ```

   where *N* is the value of `OF(L type)`, followed by either:

   1. `SET IND(LFPT)type = value` or
   2. `SETSW IND(LFPT)SW = value`

   depending on *type*.

*Notes*

a. In most L-maps it should be possible to optimise multiple stacking operations by bumping the stack pointer only once for the whole set of value(s) to be stacked. However the `STACK` statement occurs relatively infrequently so there is no need to worry too much about optimisation.
b. The values must be stacked in the order specified.

### 4.5.8 The TEST Statement

*Purpose*

Multi-way branch statement.

*General Form*

```
TEST V-NMSW GOING program label <<, program label *>>
```

*Structure Representation*

```
TEST TYPE GOING N1 OPT , N1 OR NL ALL
```

*Example*

```
TEST TYPE GOING SKIP, INS, WARN
```

*Restrictions*

The value of `%A1.` lies between 0 and `%T1-2.` (inclusive).

*Action*

The action is equivalent to:

```
IF %A1. = 0 THEN GO TO %A2.
IF %A1. = 1 THEN GO TO %A3.
. . .
IF %A1. = %T1-2. THEN GO TO %AT1.
```

### 4.5.9 The `UNSTACK` Statement

*Purpose*

Unstacks individual variables.

*General Form*

```
UNSTACK << V-NMPT ( (NM) ) *>> FROM BSTACK
                    (PT)
```

*Structure Representation*

```
UNSTACK N1 OPT ( ) N1 OR FROM WITHS BSTACK WITHS NL ALL
```

*Example*

```
UNSTACK DELPT(PT) MTCHPT(PT) MCHLIN(NM) FROM BSTACK
```

*Restrictions*

The parenthesised letters indicate the type of the preceding variable. The backwards stack pointer (`LFPT`) never occurs as an argument to `UNSTACK`.

*Action*

For each variable, starting with the first and ending with the last, the action should be:

```
SET variable = IND (LFPT) (NM)
                          (PT)
SET LFPT = LFPT + OF( (LNM) )
                      (LPT)
```

# 5 Declarative Statements and Initial Values

The `VARS` SECTION contains all the declarative statements in L. Every variable is declared exactly once. All variables have global scope.

A glance at the `VARS` SECTION will show that some groups of declarations are organised into `BLOCK`s. This means that all the variables should be allocated contiguous storage (although any or all of them can be aligned as would be required, for instance, on System/360). This grouping is to allow the variables to be moved about as a single unit using the block moving statements of Section 4.3 [Block Moving Statements], page 25. For example all the variables describing the state of scan in ML/I are grouped into a `BLOCK` so that the whole lot can conveniently be stacked when a macro call is processed. Corresponding to each `BLOCK` is a constant-defining macro denoting its size (see Section 3.3.4.4 [Number Constants], page 14).

A block is written thus:

```
BLOCKDEC identifier, subsidiary comment

<< declarative statement *>>

ENDBLOCK identifier
```

The same identifier follows both `BLOCKDEC` and `ENDBLOCK` and acts as the name of the block. Block names occur as arguments to the `BLOCK` macro described in Section 4.3 [Block Moving Statements], page 25. The declarative statements may be `DEC` statements or (in one case only in the current logic of ML/I) nested block declarations.

If `BLOCKDEC` and `ENDBLOCK` are regarded as separate macros, their structure representations could be:

a. `BLOCKDEC , NL`
b. `ENDBLOCK NL`

## 5.1 Static and Dynamic Initialisation

Depending on the environment in which ML/I is to be run, program variables may be initialised statically (i.e. the initial value is loaded into the storage occupied by the variable) or dynamically (i.e. instructions are executed at the start of each ML/I process to set up the initial values). Dynamic initialisation obviates the need to reload ML/I between one process and the next.

In order that either kind of initialisation can be performed, the logic of ML/I contains initialisation information in two places, viz:

a. As an argument to `DEC`, the declarative statement.
b. In a `SECTION` of the logic called `INVALS`, which contains a series of assignment statements.

In each L-map one of these will be ignored and the other will cause code to be generated. The `INVALS` SECTION can be deleted by the skip:

```
MCSKIP SECTION WITHS INVALS ENDSECT WITHS INVALS
```

Each assignment statement (`SET` or `SETSW`) in the `INVALS SECTION` is preceded by the comment `/- IN -/` to allow special action to be taken. For example if all storage is initially zeroised statements of form:

```
/- IN -/ SET XXX = 0
```

could be ignored.

## 5.2 Declarative Statements for Single Variables

The two declarative statements used in L to declare individual variables are `DEC` and `EQUATE`. These are described below.

### 5.2.1 The `DEC` Statement

*Purpose*

Declares and reserves storage for a single variable.

*General Form*

`DEC` *`V-NMPTSW`* `<< INIT` *`C-NMPTSW`* `?>>`

*Structure Representation*

```
DEC OPT INIT N1 OR N1 NL ALL
```

*Examples*

a. `DEC INSW`
b. `DEC GHSHPT INIT AD(GHSHTB)PT`

*Restrictions*

If there is an initial value for a variable it is of the same data type as the variable.

*Action*

Reserve storage for the variable. If static initialisation is to be performed then the initial value, if any, should be placed in the storage reserved.

### 5.2.2 The `EQUATE` Statement

*Purpose*

Sharing of storage between two variables.

*General Form*

`EQUATE` *`V-NMPTSW`* `TO` *`V-NMPTSW`*

*Structure Representation*

```
EQUATE TO NL
```

*Example*

```
EQUATE WNIDPT TO ERIAPT
```

*Restrictions*

`%A1.` and `%A2.` are of the same data type. `%A2.` has been previously declared using the `DEC` statement.

*Action*

The storage for `%A1.` is made coincident with that for `%A2.` (alternatively if this is, for some reason, undesirable, then `%A1.` may be given storage of its own exactly as if it had been an argument to the `DEC` statement).

# 6 The Data SECTIONs

The data SECTIONs contain the descriptions of the data required by the logic of ML/I. This consists of keywords, various tables and the names of the operation macros and their delimiters.

There are four special statements and two special constant-defining macros needed in the data SECTIONs. These are described below. Labels occur in the data SECTIONs but these data labels always precede the `DC` statement. No alignment should be performed in the data SECTIONs, i.e. in machines such as IBM System/360 where some data is often aligned to word boundaries, this alignment should be suppressed.

## 6.1 Constant-Defining Macros for Data

Descriptions of the two extra constant-defining macros used exclusively in the data SECTIONs follow.

### 6.1.1 The RL Macro

*Purpose*

Specifies a relative link.

*General Form*

```
RL ( data label << (+) C-NM *>> )
                   (-)
```

*Structure Representation*

```
RL WITHS ( )
```

*Examples*

a. `RL(DUNLES)`
b. `RL(DSEMIC + OF(LNM))`

*Restrictions*

`RL` occurs only as an argument to the `DC` or `OPMAC` statements.

*Action*

Generate numerical constant having value

```
address specified by %A1. - address occupied
                            by constant itself
```

*Notes*

If the object language is an assembly language where the location counter is specified by *, the replacement text of `RL` might be

```
%A1. - *
```

In the current logic of ML/I the resultant constant is always positive.

### 6.1.2 The `LID` Macro

*Purpose*

Specifies a data field consisting of a character constant preceded by its length.

*General Form*

`LID[<<` ***`character`*** `*>>]`

*Structure Representation*

`LID WITHS [ ]`

`LID` should be a straight-scan macro.

*Example*

`LID[OPT]`

*Restrictions*

`LID` occurs only as an argument to the `DC` statement. The restrictions on `LID` otherwise are the same as apply to an occurrence of the quote macro in the data SECTIONs.

*Action*

Generate a constant consisting of a number representing the length of the argument (i.e. `MCLENG(%WB1.)`) multiplied by `OF(LCH)` followed by a character string constant representing the argument.

*Notes*

The `LID` macro might need to call the quote macro, which is a straight-scan macro. If so the call could be performed thus:

```
MCDEF TMP AS <'>%WB1.<'>
TMP
```

## 6.2 Data-Defining Statements

Descriptions of the statements for defining data follow.

### 6.2.1 The `DC` Statement

*Purpose*

Specifies a list of constants.

*General Form*

`DC` ***`argument`*** `<<,` ***`argument`*** `*?>>`

*Structure Representation*

`DC N1 OPT , N1 OR NL ALL`

*Example*

`DC 1,LID(XYZ),ENDCHN,RL(KOR-OF(LNM))`

*Restrictions*

Each argument is one of the following:

a. a call of the `RL` or `LID` macros.
b. a call of the quote macro.
c. a constant of type number. This may be specified literally or by means of a constant-defining macro.

*Action*

Generate sequence of data constants as specified by the list of arguments.

*Notes*

`DC` may be labelled and, if so, the label should refer to the first data constant generated.

### 6.2.2 The `LAYCHAIN` Statement

*Purpose*

Specifies the structure representation keywords for layout characters.

*General Form*

`LAYCHAIN`

*Structure Representation*

`LAYCHAIN NL`

*Restrictions*

`LAYCHAIN` only occurs once in the logic of ML/I.

*Action*

`LAYCHAIN` is replaced by specifications of the keywords to be used in structure representations to represent layout characters. Each specification should be represented by:

```
DC RL(next), RL(rep), LID(name), ENDCHN
```

where *name* is the name of the keyword, *rep* is the address of the character represented by the keyword (this is supplied by the `HETABLES` statement, which is described later) and *next* is the address of the next keyword specification. In the case of the last specification *next* is `KSPACS`.

*Notes*

a. For reasons of compatibility the following standard names are to be preferred when applicable:

| | |
|---|---|
| `SPACE` | representing space. |
| `NL` | representing newline. |
| `SL` | representing startline. |
| `TAB` | representing tab. |

b. The keyword `SPACES` forms part of the main logic and is not supplied by the `LAYCHAIN` statement. `LAYCHAIN` should always supply a keyword to represent a single space.

c. As an example of `LAYCHAIN`, if the two keywords `SPACE` and `NL` are required, then `LAYCHAIN` should be replaced by:

```
               DC RL(KNL),RL(RSPAC),LID[SPACE],ENDCHN
        [KNL]   DC RL(KSPACS),RL(RNL),LID[NL],ENDCHN
```

where the following is supplied as part of the `HETABLES` statement:

```
        [RSPAC] DC ' '
        [RNL]   DC '$'
```

### 6.2.3 Hash-Tables and their Definition

In order to speed up the recognition of construction names, the logic of ML/I uses a hashing technique. This involves the use of tables of pointers called *hash-tables*. The size of these tables, which is machine-dependent, is given by the constant defining macro `LHV`. Typical values of `LHV` are `16*LPT` for a machine with 8K words, `32*LPT` for a machine with 16K words and `64*LPT` for a larger machine, where `LPT` is the amount of storage occupied by a pointer.

Each implementation has its own hashing function (see Section 7.1.2 [The `MDFIND` Subroutine], page 47) which maps the set of all possible atoms evenly into the values:

```
0, LPT, 2*LPT, 3*LPT, ..., LHV-LPT
```

Each hash-table entry is the head of a (possibly null) chain of pointers, which joins together all construction names whose first atom hashes to the number given by the offset of the hash-table entry from the start of the hash-table. Thus the first hash-table entry represents value `0`, the second `LPT`, and so on. When an atom of text is scanned it is mapped into a number by the hashing function and then compared with all the construction names on the relevant hash chain.

When implementing ML/I it is necessary to define a hash-table that reflects the initial state of the environment (in early implementations of ML/I there were two such hash-tables, one global and one local, but these have now been combined). For most implementations the initial state of the environment will consist only of the operation macros. These need to be connected together on the relevant chains, with the chain heads in the initial hash-table.

#### 6.2.3.1 The `HETABLES` Statement

*Purpose*

Specifies miscellaneous pieces of data.

*General Form*

```
HETABLES
```

*Structure Representation*

```
HETABLES NL
```

*Restrictions*

`HETABLES` only occurs once in the logic of ML/I.

*Action*

a. Reserve storage and set initial values for the two following tables (note that the 'Size' value is expressed in terms of constant-defining macros):

| **Name** | **Size** | **Initial Values** |
|---|---|---|
| `ERBLOC` | `EDBSZ` | None |
| `GHSHTB` | `LHV + 4*LPT + LSW` | See below |

In each case the name should be attached to the start of the table (the names are used as arguments to the `AD` macro). The initial value of `GHSHTB` should be as follows:

| **Number and type** | **Value** |
|---|---|
| `LHV/LPT` pointers | Heads of hash chains |
| 4 pointers | Any value greater than or equal to the highest possible value for a pointer. |
| 1 switch | 7 |

The `GHSHTB` table (as distinct from the values of the S-variables — see below) is never changed during the running of ML/I and therefore does not need to be reset if ML/I is restarted without being reloaded.

b. Define the layout characters required by the `LAYCHAIN` statement. Each layout character should be preceded by a data label so it can be referenced by the corresponding keyword. All these layout character representations must be supplied explicitly by `HETABLES` and no attempt must be made to "common them up" with representations lying elsewhere in storage.

c. Reserve storage for the S-variables. Let N be the number of S-variables for the implementation (N should not be less than 9). Then N+1 numbers should be reserved. The last number should contain the value of N and should be labelled `SVEC`. The remaining numbers contain the values of the S-variables in inverse order, i.e. the first is `S(N)` and the last `S(1)`. These should be given appropriate initial values, either statically or dynamically. `S1` to `S9` should be initially zero. Thus for example if N was 10 the S-variables might be set up thus:

```
        DC 0,0,0,0,0,0,0,0,0,0
[SVEC]  DC 10
```

### 6.2.4 The `OPMAC` Statement

*Purpose*

Specifies data representing operation macro.

*General Form*

`OPMAC` *`C-CH`* `<< +` *`C-CH`* `?>> ,` *`C-NM,C-NM,C-NM`*

*Structure Representation*

```
OPMAC OPT + N1 OR N1 , ALL , , NL
```

*Examples*

a. `OPMAC 'MCNODEF',ENDCHN,LOCMK,1`

b. `OPMAC 'MCSUB'+'(',RL(DCOM1),OPMK,14`

*Action*

Generate series of constants representing the operation macro and its attributes. These constants are as follows:

| | **Type** | **Value** |
|---|---|---|
| 1) | Pointer | Hash chain pointer. |
| 2) | Number | Value of `ENDCHN` constant-defining macro. |
| 3) | Number | Length of macro name (i.e. `MCLENG(%WA1.-2)` multiplied by `OF(LCH)`). |
| 4) | Characters | Character string represented by `%WA1.` |
| 4a) | Number | Value of `WTHSMK` constant-defining macro. |
| 4b) | Number | Length of second atom of macro name (i.e. `MCLENG(%WA2.)-2` multiplied by `OF(LCH).` |
| 4c) | Characters | Character string represented by `%WA2.` |
| 5) | Number | Value of `%AT1-2.` |
| 6) | Switch | Has value 1 (denotes operation macro). |
| 7) | Number | Value of `%AT1-1.` |
| 8) | Number | Value of `%AT1.` |

*Notes*

a. Items 4a), 4b) and 4c) will be present only if the additional argument (introduced by `+`) is present.

b. The hash chain pointer will normally need to be filled in by hand. Furthermore the result of each `OPMAC` will normally need to be given a created label in order that a hash pointer can reference it.

c. The last argument is a decimal number. This is the only place in the logic of ML/I where a number greater than 7 is specified literally (as distinct from by means of a constant-defining macro). Special action may be necessary with this argument if the object language works in, say, octal or hexadecimal.

# 7 The MD-logic

The MD-logic of ML/I consists of a number of pieces of machine-dependent code each of which is either represented as a subroutine or designated by a label. The complete list of subroutines is: `MDCONV`, `MDERPR`, `MDFIND`, `MDINIT`, `MDNUM`, `MDOP`, `MDQUOT` and `MDTEST`, and the complete list of labels is `MDABRT`, `MDGOBC` and `MDHALT`, together with some initialisation code.

With the exception of `MDTEST` and `MDFIND` these pieces of code are not heavily used and they may be written to be as concise as possible rather than as fast as possible. They should use their own variables for intermediate working and should not clobber any variables used in the MI-logic unless this is explicitly allowed.

The pieces of code are grouped under four categories, which will be discussed in the order below:

a. Code dependent on internal representation.
b. Subroutines for error messages.
c. Initialisation and finalisation.
d. The `MDOP` subroutine.

## 7.1 Code Dependent on Internal Representations

This Section describes those parts of the logic of ML/I that are dependent on the internal representation of characters and numbers on the object machine. If there is any choice in the representation of characters then the following points should be borne in mind:

a. It should be easy to tell whether a character is a letter and whether it is a digit.
b. It should be easy to convert from the character representation of a digit to its representation as a number.

On machines where numbers are represented as a string of characters rather than in binary, consideration b) above will not apply and the routines for converting between character and numerical representations will be very short or even null.

Descriptions of the pieces of machine-dependent code follow.

### 7.1.1 The `MDTEST` Subroutine

*Description*

Subroutine with both parameter and exit label (however, see Note below). Parameter is a pointer.

*Action*

If the character pointed at by the parameter is not a letter or a digit then go to the exit label. Otherwise return.

*Note*

It is highly desirable that `MDTEST` be replaced by in-line code in the MI-logic rather than act as a subroutine. To generate in-line code a macro should be defined with structure representation:

```
CALL WITHS MDTEST WITHS ( ) WITHS PT WITHS EXIT NL
```

### 7.1.2 The `MDFIND` Subroutine

*Description*

Subroutine with no parameter or exit label. `MDFIND` is the hashing function. The atom to be hashed is described by `IDPT` (which points to its first character), `SPT` (which points at its last character) and `IDLEN` (the number of characters in the atom).

*Action*

Map the atom into a number as described in Section 6.2.3 [Hash-Tables and their Definition], page 43 and set `OFFSET` as the value of this number. Hence `OFFSET` should be a multiple of `LPT` and should satisfy the relation:

```
0 <= OFFSET < LHV
```

Also set `HTABPT` equal to `HASHPT + OFFSET`.

### 7.1.3 The `MDCONV` Subroutine

*Description*

Subroutine with no parameter or exit label. `MDCONV` converts a number to a character string representation. The number to be converted, which may be positive, negative or zero, is given by `MEVAL`.

*Clobberable*

`MEVAL`.

*Action*

Convert the value of `MEVAL` (as a decimal number) to a character string representation (which should be stored in some workspace reserved for use by the `MDCONV` subroutine). Set `IDPT` to point at this character string and set `IDLEN` as the number of characters in it. The character string should consist only of digits except that it should be preceded by a minus sign if `MEVAL` is negative. It should contain no redundant leading spaces.

### 7.1.4 The `MDNUM` Subroutine

*Description*

Subroutine with no exit label. `MDNUM` converts from the character representation of a number to its internal form. This number is a non-negative decimal integer. `IDPT` points at the first character of the string to be converted and `SPT` points at the last character (`SPT >= IDPT`).

*Clobberable*

`MEVAL` (but *not* `IDPT`).

*Action*

If the character at `IDPT` is not a digit then go to the exit label. Otherwise, if any character between `IDPT` and `SPT` (inclusive) is not a digit then go to `ERLIA`; if all are digits then set `MEVAL` to the value of the decimal number represented by the string of digits and return.

### 7.1.5 The `MDGOBC` Label

*Description*

Label. `MDGOBC` is used when the `BC` delimiter occurs on a call of `MCGO`. Before `MDGOBC` is gone to the following work is done:

1. *arg B* of the call of `MCGO` is evaluated, `INFFPT` is set to point at the first character of its value and `ERIAPT` is set to point at the character position beyond the last character.
2. *arg C* of the call of `MCGO` is evaluated and it is verified that its value consists of a single character. `IDPT` is set to point at this character.

*Clobberable*

`IDPT`, `IDLEN`.

*Action*

Go to one of the following three labels, depending on the form of *arg B* and *arg C*:

| | | |
|---|---|---|
| 1) | `ERLIA` | *arg C* is illegal. |
| 2) | `GOSUC` | *arg B* belongs to the class designated by *arg C*. |
| 3) | `GOFAIL` | *arg B* does not belong to the class designated by *arg C*. |

*Action*

a. In current implementations *arg C* can be `L` (for letter), `N` (for number) or `I` (for identifier) but this list can be extended if desired.

b. `MDGOBC` requires careful coding, due to various special cases. In particular *arg B* may be null, in which case the condition should fail. Furthermore such strings as '`+ 1`' and '`+ - 1`' should be accepted as numbers but such strings as '`+`', '`+-`', '`+ 1 -`' and '`+ 1 - 2`' should not.

## 7.2 Subroutines for Error Messages

The two machine dependent subroutines `MDERPR` and `MDQUOT` are used in the production of error messages from ML/I. These two subroutines and the `PRTEXT` statement (see Section 4.4.3 [The `PRTEXT` Statement], page 30) are the only parts of the logic concerned with the production of error messages.

### 7.2.1 The `MDERPR` Subroutine

*Description*

Subroutine with no parameter or exit label. `MDERPR` is the main printing routine. The piece of text to be printed is pointed at by `IDPT`. `IDLEN` gives the number of characters in the text.

*Clobberable*

`IDPT`.

*Action*

Print the text on the error message listing. Note that the text may be null (in which case `IDLEN` is zero). Startline characters should be replaced by the characters `(SL)` to make their presence obvious to a reader.

### 7.2.2 The `MDQUOT` Subroutine

*Description*

Subroutine with no parameter or exit label.

*Action*

Print a quote on the error message listing (`MDQUOT` has been made a machine-dependent routine for pragmatic reasons in that the printing of a quote is so often a special case).

## 7.3 Initialisation and Finalisation

It is necessary to code by hand some logic to provide an environment in which ML/I is to operate. This code performs the necessary initialisation before the MI-logic of ML/I is entered and clears up when it has finished.

### 7.3.1 Initialisation Code

It is necessary to write a certain amount of code for initialisation which is to be executed immediately after ML/I is loaded and immediately before the MI-logic is entered.

One of the functions of this code is to process *control statements* supplied by the user. Typical functions of such statements would be:

a. To describe the form of the input and output.
b. To specify the size of the workspace.
c. To specify whether a print-out of all construction names is required (see Section 7.3.3 [The `MDHALT` Label], page 51).

The format and the physical form of these control statements will vary considerably between implementations.

A second function of the initialisation code is to set up variables to describe the stacks and the initial state of the environment. Normally the initial environment will consist only of the operation macros but it is possible for the initialisation routine to bring in pre-defined local or global substitution macros and other constructions. For instance a certain control statement might be made to cause a particular package of definitions to be included. If new definitions are included they must be added to the relevant hash chains. Global definitions should be stored at the start of the forwards stack and local definitions at the start of the backwards stack.

The initial state of the stacks and the pointers that describe them is illustrated by the following diagram:

```
Start of workspace --->  +----------------------+
                         | pre-defined global   |
                         | definitions (if any) |
                         +----------------------+
                         | permanent variables  |
             PVARPT ---> +----------------------+
                         |      free space      |
                         |                      |
                         \/\/\/\/\/\/\/\/\/\/\/\/

                         \/\/\/\/\/\/\/\/\/\/\/\/
                         |                      |
                         |      free space      |
               LFPT ---> +----------------------+
                         | pre-defined local    |
                         | definitions (if any) |
             ENDPT --->  +----------------------+
```

The variables that describe the initial state of the environment and the stacks are declared as a block in the MI-logic. This block is called `DIB`, meaning “Dynamic Initialisation Block”. Variables in `DIB` should be initialised as follows:

| | | |
|---|---|---|
| a. | `PVNUM` | The number of permanent variables that is allocated at the start of processing. |
| b. | `GLBWSW` | Set to value 6 if the predefined global definitions include a warning marker and to value 7 otherwise (the switch within `HASHTB` should be initialised in a similar way if there is a local warning marker, see the `HETABLES` statement of Section 6.2.3.1 [The `HETABLES` Statement], page 43). |
| c. | `PVARPT` | The address given by: *address of start of workspace* `+` *size of predefined global definitions* `+ PVNUM * OF(LNM)` |
| d. | `ENDPT` | Pointer to the address beyond the end of workspace. |
| e. | `LFPT` | The address given by: `ENDPT -` *size of predefined local definitions* |

Lastly the initialisation code should preset the I/O, initialise the S-variables if this is to be done dynamically, perform any further initialisation peculiar to the implementation and then branch to the label `BEGIN` in the MI-logic in order to start ML/I executing (if the `INVALS` SECTION has been deleted from the MI-logic then a branch should be made to the label `MBEGIN` rather than the label `BEGIN`. If, as occurred on one L-map, the mapping of the `VARS` SECTION yields some executable initialisation code, then the flow of control should be adjusted to include this code).

### 7.3.2 The `MDINIT` Subroutine

*Description*

Subroutine with no parameter or exit label. `MDINIT` is called when all initialisation in both the MD-logic and the MI-logic has been performed and ML/I is hence ready to start processing.

*Action*

In most implementations `MDINIT` will be null. However one example of its use is illustrated by the PDP-7 implementation which, when `MDINIT` is called, communicates with the user and allows him to type in source text from the keyboard.

### 7.3.3 The `MDHALT` Label

*Description*

Label. The MI-logic transfers control to `MDHALT` at the end of processing.

*Action*

Print message to user that macro-processing is complete, perhaps with extra information as to how many macro calls have been performed (as given by the variable `INVOCT`) and how many source lines have been scanned.

In addition an implementation may print out the list of all names in the environment at this stage (this print-out may be made dependent on an option set by the user). If a print-out is desired this can be achieved by calling the subroutine `PRENV`, which lies in the MI-logic of ML/I. If, on the other hand, it is desired to omit this facility altogether from an implementation then `SECTION ENVPR` of the MI-logic, which contains the statements that implement it, can be deleted.

When `MDHALT` has finished its printing it should terminate all the I/O and, as appropriate, either halt, return control to the supervisor or continue with the next job step.

### 7.3.4 The `MDABRT` Label

*Description*

Label. The MI-logic branches to `MDABRT` if a process is aborted due to stack overflow or a system error. The branch occurs after the relevant error message has been printed.

*Action*

`MDABRT` will be coincident with `MDHALT` on most implementations. The only difference is that if macro processing has been aborted it may be considered desirable to stop the entire job rather than pass control to the next job step.

## 7.4 The `MDOP` Subroutine

*Description*

Subroutine with no parameter or exit label. `MDOP` is called from the `GETEXP` subroutine and deals with the multiplication and division operators in the calculation of macro expressions.

*Action*

Assuming the `*` and `/` operators were allowed in expressions in L, `MDOP` would be:

```
SUBROUTINE MDOP
  IF OPSW = 1 THEN
    //MULTIPLY CASE//
    SET MEVAL = OP1 * MEVAL
    // If desired, a test for overflow can be performed
       and the action ‘GO TO ERLOVF’ performed if it is
       detected. //
    RETURN FROM MDOP
  END
  //DIVIDE CASE//
  IF MEVAL = 0 THEN GO TO ERLOVF
  SET MEVAL = OP1/MEVAL
  // An overflow test can be performed, if desired. //
  RETURN FROM MDOP
ENDSUB
```

# 8 Notes on the Overall Organisation of an L-map

Experience so far has shown that it is highly desirable to split an L-map up into several passes, although this is not usually logically necessary. Furthermore it has been found that since the data SECTIONs are relatively short and involve relatively complicated complicated macros, it is quicker, at least in the short run, to encode them by hand unless the implementor is very practised in using ML/I.

An implementor mapping to an assembly language may find that it is convenient to organise his L-map into three passes in approximately the following way:

a. *Pass 1.* Deal with comments and layout statements, the `OF` macro, and constants represented by identifiers. Possibly deal with the `AD`, `BLOCK` and quote macros, though it may sometimes be more convenient to build these into the Pass 3 macros. Delete tabs from the source text. Perform preprocessing of `IF` statements and any other statements presenting difficulties in later passes.
b. *Pass 2.* Map all statements into *pseudo-instructions*. Pseudo-instructions will be like instructions for the object machine except that the operands will be more general. Typically there might be a pseudo-instruction to evaluate an expression and place its value in a register, which, say, had the form:

   LOADEX *register, arithmetic expression*

   In addition to `LOADEX` there might be a pseudo-instruction of form:

   INST *op code register, VCI-CHNMPTSW*

   which dealt with indirect addresses as operands and turned constants into literals. If a machine has, like IBM System/360, several instruction formats there may need to be several different `INST` pseudo-instructions.
c. *Pass 3.* Deal with labels and turn pseudo-instructions into pure machine instructions.

In addition there might be a fourth pass to perform optimisation.

There are two small precautions, both concerned with bracketing, that need to be taken on a multi-pass L-map. Firstly if `IND` is processed on Pass 3 and `CALL` on Pass 2 then in the statement

```
CALL SUB ( IND ( IDPT ) NM ) NM
```

the first right parenthesis will be taken as the delimiter of `CALL` on Pass 2. To cause correct matching of parentheses the skip:

```
MCSKIP MDT, IND WITHS ( )
```

is necessary. Similar action may be necessary for other uses of parentheses, for example the `AD` and `BLOCK` macros and parentheses introduced as the result of mapping macros.

Secondly if labels are processed before `LID` and `PRTEXT`, the skips

```
MCSKIP DT, <PRTEXT WITHS[ ]>
MCSKIP DT, <LID WITHS [ ]>
```

will be necessary to avoid the brackets within those macros being taken to mean the placing of a label.

## 8.1 Scope of Mapping Macros

When performing an L-map it will be found that many mapping macros only occur in restricted contexts. For instance the mapping macros for declarative statements are only required in the `VARS` SECTION and there are instances of macros that can only occur within some other macro. However names have been chosen so that macro names are unique so it is possible to apply all mapping macros to the entire MI-logic. This is usually easier and quicker than defining macros with limited scope. The only element of care necessary is to avoid macro replacement within comments and character string constants by defining them as skips or straight-scan macros.

## 8.2 Debugging of Mapping Macros

The debugging of mapping macros should be performed using specially written test statements rather than the logic of ML/I itself. There is a skeleton version of the logic of ML/I which is also useful for testing. This skeleton version does little more than read in an atom and output it again but it is a good idea to map it into the object language as a preliminary to mapping the full MI-logic. When this skeleton version is working on the object machine it can be used to test the I/O and other hand-coded material.

# Appendix A Statement Prefixes

Currently the list of statement prefixes, as described in Section 2.5 [Comments], page 7, in the logic of ML/I is as follows:

a. The prefix `/- OVP -/` is used to precede `SET` statements where arithmetic overflow is possible. See Section 3.1 [Data types], page 9.
b. The prefix `/- IN -/` is used to precede all `SET` and `SETSW` statements in the `INVALS SECTION`. See Section 5.1 [Static and Dynamic Initialisation], page 37.
c. The prefix `/- CSS -/` is used to precede any label that is gone to from within a subroutine. See Section 4.1.1.1 [The `SUBROUTINE` statement], page 17.

# Statement/Macro Index

# MD-logic Index

# Concept Index