Implementing software using the LOWL language

Supplement 3: ML/I

Second Edition

August 2015

P.J. Brown, R.D. Eager

Table of Contents

Preface to the Second Edition

This Edition has been rewritten in Texinfo, so that it can be published in both printed and machine readable form; this has necessitated some re-wording and re-ordering of the text.

There have also been one or two corrections of minor errors in the previous edition, as well as a few clarifications. The text has been modernised in places, to reflect current technology, and the Section on `MDREAD` has been expanded.

1 LOWL extensions

Before the extensions to LOWL needed by ML/I are described, it is necessary to explain its *hash table*, since this affects several of the extensions.

1.1 The hash table

Clearly, ML/I would be very slow if it compared each atom of scanned text with all possible construction names. Hence a hashing algorithm is employed. This involves a hash table of (typically) 32 pointers, each of which is the head of a chain of construction names. A zero pointer means the end of a chain. The implementor chooses a hashing algorithm (see Section 2.5 [The `MDFIND` subroutine], page 7) which assigns every possible construction name to a hash chain.

Every time a new construction is defined, ML/I calls `MDFIND` to find out which hash chain it should be added to. However, the names of the operation macros are built into ML/I in its fixed tables, and the initial state of the hash table and the chains connecting the built-in macros need to be supplied as constants. There are two statements (`HASH` and `THASH`) to generate these constants but it is usually very hard to write macros to work out hash chains, and, since there are so few of them, it is often best done by hand.

The size of the hash table need not be 32 pointers. For a small machine it can be reduced to 16 pointers and for a large one increased to 64 pointers or even 128 (a power of two is not essential, but is usually more efficient in terms of the actual hashing algorithm).

The following diagram shows the hash chains where `MCDEF` and `MCINS` are both on chain 1, and `MCLENG` is on chain 3.

```
         +-----------+
Chain 0  |     0     |
         +-----------+                   +-----+     +-----+
Chain 1  |       ----+------------------>|   --+---->|  0  |
         +-----------+                   +-----+     +-----+
Chain 2  |     0     |                      M           M
         +-----------+     +-----+          C           C
Chain 3  |       ----+---->|  0  |          D           I
         +-----------+     +-----+          E           N
         |           |        M             F           S
         .           .        C
         .           .        L
         .           .        E
         .           .        N
         .           .        G
         .           .
```

When ML/I is running, it may actually create several extra hash tables, but the implementor need not be concerned with these.

1.2 Extensions to LOWL

ML/I requires the following extensions to LOWL.

1.2.1 Subsidiary macro of `OF`

`LHV` is a subsidiary macro of `OF` which gives the size of the hash table. Thus if the hash table consisted of 32 pointers, `LHV` would equal 32 times `LNM`.

1.2.2 Expression extension in OF

The `OF` macro requires one additional form of expression to be implemented. This is of the form

```
N*S+N*S
```

This form is used only within the second argument to the `RL` macro; this is not often used, so many implementors will not have to worry about this.

1.2.3 Character constants

Two extra names for representing character constants are used. These are `STOPCD` and `SLREP`, and should each be assigned a unique code that cannot occur naturally in the source text.

1.2.4 Clobbering of table items

The `MCALTER` operation macro in ML/I has the effect of changing table items. In any multi-user application where the tables are shared, or in any implementation where these tables are conveniently kept in read-only storage, `MCALTER` should be forbidden. This is easily done by making the `HASH` statement (see Section 1.2.5 [Statements for defining table items], page 3) leave it off the hash chains.

1.2.5 Statements for defining table items

The following additional statements are needed:

a.

```
RL      table label,OF
```

defines an item containing the value of the relative offset of the given table label. The second, subsidiary argument gives the value of the offset. For an assembler where the character `.` represented the location counter, then

```
RL      PIG,OF(3*LNM+6*LCH)
```

might be represented as

```
PIG - .
```

On less powerful assemblers it is often necessary to make use of the subsidiary argument.

b.

```
HASH    'characters'
```

defines an item containing the hash chain link for the macro named within the quote signs.

c.

```
THASH
```

defines the set of table items representing the starts of all hash chains. `THASH` only occurs once, and this is after all the occurrences of `HASH`.

d.

```
          WTHS
```

means a table item consisting of a unique numerical marker. This should be the highest integer acceptable to the object machine.

1.2.6 The ORL statement

This additional statement is needed for ML/I:

`ORL`	*N*	inclusive "or" A with literal value *N*.

1.2.7 The linkroutine

There exists a special routine called the *linkroutine*. Its name is `STKARG`, and it is essentially a recursive subroutine. However, the MI-logic of ML/I takes care of all problems connected with recursion and the task of the implementor is therefore very simple. The linkroutine is declared by the statement:

```
        LINKR   STKARG
```

This should be mapped exactly as a `SUBR` with no parameter except that the return link should be placed in the variable `LINKPT` and not in a place designated by the implementor (`LINKPT` is automatically stacked and unstacked in recursive situations). The return from the linkroutine is written simply:

```
        LINKB
```

which should be mapped into a jump to the return address contained in `LINKPT`.

`STKARG` is called by an ordinary `GOSUB` statement. However the linkroutine is best treated as part of the main logic, rather than as a subroutine, as there are branches between the linkroutine and the main logic. If subroutine return addresses are being kept on a stack, the return link to the linkroutine should *not* be placed on the stack.

2 The MD-logic

2.1 I/O options

Basically ML/I requires three input/output streams:

a. the input stream, which contains the source text.
b. the messages stream, which contains error and informational messages. This is referred to as the *debugging file* in the *ML/I User's Manual*.
c. the results stream, which contains the output text. In cases where ML/I is being used as a preprocessor, the results stream may often be a temporary file that is subsequently used as input by a compiler or assembler.

If the implementation of ML/I is intended for production work, however, these facilities may need to be supplemented by some extra options, for example listings of the source text and/or output text. Being implementation dependent, these options need to be provided by the MD-logic. Sometimes an operating system may automatically provide all the required options, but more often it has proved necessary for the implementor to program some of them into the MD-logic routines. Options that are open to the user can either be provided by the S-variables from `S10` upwards (with initial values reflecting the default options) or by parameters to the command that invokes ML/I. A combination of both is ideal; S-variables are initially set to default options, but parameters can cause these to be changed; further dynamic changes can be made by assignments to the S-variables during a process.

The following list shows a fairly comprehensive collection of options that might be provided by means of S-variables. Ideally all the options should be dynamically changeable. The implementor is certainly not expected to follow this scheme exactly, but it gives a general guide to what may be provided.

2.1.1 Input controls

`S10`	input channel number.
`S11`	starting column for line-based input.
`S12`	terminating column for line-based input.
`S13`	internal code of character that terminates input. Often this option is not relevant as the operating system has its own conventions.
`S14`	controls whether source is to be listed.
`S15`	controls whether spaces at the end of input lines are to be deleted.
`S16`	Used to control character code translation. Characters with the code given by `S16` are translated to characters with the code given by `S17`, on input. Initially `S16` could be -1, so no translations are performed. This allows characters to be input from devices that do not directly support them.
`S17`	See `S16` above.
`S18`	controls input formatting options.
`S19`	controls input formatting options.

2.1.2 Output controls

`S20`	controls whether output is to be listed, and if so whether a line count is to be supplied.
`S21`	on/off option for main output channel (so that output can be wholly or partially suppressed).
`S22`	on/off option for secondary output channel.
`S23`	on/off option for tertiary output channel.
`S24`	control for output formatting options.
`S25`	more output formatting options.

2.1.3 Facts about system

`S30`	might show which operating system is in use, how much memory is available, etc.

2.2 Initialisation code

The initialisation code should perform the following actions:

1. Initialise the S-variables. S-variables must be stored in reverse order (i.e. `S1` is last) and be immediately followed by a numerical field containing the number of S-variables. The variable `SVARPT` should be set to point at this number. There must be at least nine S-variables. `S1` to `S9` should be initialised to zero and the remaining S-variables should be initialised to reflect default options. The following pictorial example shows how the S-variables might be set if there were eleven of them, and `S10` and `S11` both had an initial value of one.

```
                 1
                 1
                 0
                 0
                 0
                 0
                 0
                 0
                 0
                 0
                 0
     SVARPT ----> 11
```

2. Deal with any parameters to the call of ML/I. In an interactive environment this might mean a dialogue with the user.
3. Perform the common initialisation code described in the LOWL manual. The stack size needs to be at least 500 words, and 4-6K is a suitable default size.

2.3 The `MDQUIT` subroutine

The `MDQUIT` subroutine should perform the functions described in the LOWL manual.

2.4 The `MDCONV` subroutine

`MDCONV` is a subroutine with a single exit and no parameter. It should take the value of the variable `MEVAL` and convert it to a character string representation. This representation should be a string of decimal digits, preceded by a minus sign if `MEVAL` is negative, and containing no redundant leading zeros. The string should be built up in some storage locations specially reserved for the purpose. `IDPT` should be set to point at the first character and `IDLEN` should be set as the length of the string (i.e. as `LCH` multiplied by the number of characters in the string). The original value of `MEVAL` need not be preserved.

2.5 The `MDFIND` subroutine

`MDFIND` is the hashing algorithm, and is a subroutine with one exit and no parameter. The atom to be hashed is described by `IDPT` (which points at the first character), `SPT` (which points at the last character) and `IDLEN` (the number of characters in the atom, multiplied by `LCH`). `MDFIND` should set the variable `HTABPT` to the head of the hash chain to which the atom belongs. The variable `HASHPT` points at the hash table, and therefore `HTABPT` should be assigned a value of the form `HASHPT+`*`N`* where *N*, the hash chain number, is a multiple of `LNM` and satisfies the relation:

    0 <= N < LHV

A simple hashing algorithm, though not the best one, is to add together `IDLEN` and the two characters pointed at by `IDPT` and `SPT`, multiply by `LNM` and “and” out redundant high-order bits (this is why it is useful if `LHV` is a power of two). `HTABPT` is then set to the resultant value, with `HASHPT` added to it.

2.6 The `MDOP` subroutine

`MDOP` is a subroutine to perform multiplication and division. It has a single exit label and no parameter. It should test the variable `OPSW`; if it is one it should multiply `OP1` by `MEVAL` and place the result in `MEVAL`; if `OPSW` is not one it should divide `OP1` by `MEVAL` and place the result in `MEVAL`. In the latter case there should be a prior test to see if the divisor (`MEVAL`) is zero, and if so a branch should be made to the label `ERLOVF` in the MI-logic. Other cases of overflow are best ignored.

The division operator should give an integer result. The rules for integer division are given in the *ML/I User’s Manual*; check them carefully, as they may not be quite what is normally expected.

2.7 The `MDREAD` subroutine

`MDREAD` is the input subroutine. It has two exits and no parameter.

Exit 1 should be used when the source text has become exhausted (if the last line of input received has not been terminated with a newline, the MI-logic will automatically add a newline).

Otherwise the next character of the source text should be read and placed in the C register, and exit 2 should be used. At the end of each line, a newline character should be supplied. There are various possibilities depending on the operating system; for example:

a. The operating system is record-based, with no concept of a newline character. In this case, supply a newline character at the end of each record, using the same value as for `NLREP`.
b. The operating system terminates each line with a carriage return, line feed pair. In this case, ignore carriage return and convert linefeed to `NLREP` (it is very likely that the code for line feed will be the same as for `NLREP` anyway).
c. The operating system terminates each line with a single character, e.g. line feed. In this case, convert line feed to `NLREP`; as in b), this may be a null conversion.

If an illegal character is found, an error message can be produced by calling the MI-logic subroutine `ERSIC`; after this, the *error character* (see Chapter 6 of the *ML/I User's Manual*) should be placed in C.

The `MDREAD` subroutine is responsible for providing the input options described in Section 2.1.1 [Input controls], page 5, and in particular for switching input streams if this facility is to be made available.

2.8 The `MDOUCH` subroutine

`MDOUCH` is the subroutine which outputs all the output text from ML/I. It has one exit and no parameter. Text is output character by character, and the character to be output is in the C register. The character set is exactly that allowed by the `MDREAD` subroutine; it therefore includes "newline" but not the imaginary characters represented by `SLREP` and `STOPCD`. Lines of output may be arbitrarily long, and `MDOUCH` is responsible for dealing with any overflow. It is also responsible for providing the output options described in Section 2.1.2 [Output controls], page 6, in particular multiple copies of the output or suppression of output.

2.9 The `MDERCH` subroutine

`MDERCH` is exactly similar to `MDOUCH` except that it uses the messages stream rather than the results stream.

3 Documentation

3.1 Appendices to ML/I User's Manual

For each implementation, there must be an Appendix to the *ML/I User's Manual* which describes the special features of that implementation. Each such Appendix is designated by one or more letters, and the writer should request a unique letter combination via the *Contact* link on the official ML/I web site (`http://www.ml1.org.uk`).

Most of the details given in an Appendix are dependent on the way that the MD-logic is coded; there are, however, a few details that are fixed in the LOWL version of the MI-logic of ML/I.

3.2 Example of an Appendix

There follows a sketch of how an Appendix might be written. It is useful if the section numbering remains constant, with any additional sections appearing at the end if at all possible. In this sketch, the identifying letters for the Appendix are XXX.

Implementation of ML/I on machine O

XXX.1 Restrictions and additions

This implementation of ML/I contains all the features described in the *ML/I User's Manual*, Sixth Edition.

XXX.2 Operating instructions and I/O

Describes how to use ML/I, e.g. command format and parameters, how I/O is controlled, and which options are available. Examples of typical commands are useful here.

XXX.3 Character set

Enumerates the characters that `MDREAD` *accepts. Describes any particular arrangements concerned with specific devices.*

XXX.4 Error messages

This Section is referenced in several places in Chapter 6 of the ML/I User's Manual. It should describe the debugging file (messages stream), any special messages peculiar to the implementation, the action on illegal input characters and what the error character is (if it exists). It should also contain the next paragraph.

With reference to Chapter 6 (of the *ML/I User's Manual*), the number 2N (the maximum number of characters in a piece of text inserted into an error message without being truncated) is 60. At the end of each process, a message giving processing statistics is sent to the debugging file.

XXX.5 Integer calculations

The initial environment contains ten permanent variables, all set to the value zero. All integers in, or derived from, macro expressions should be less than *(fill in as appropriate)* in magnitude. Overflow is not *(always)* detected, except in the case of division by zero, and its effect is undefined (*alter if necessary*).

XXX.6 Layout keywords

The following are the layout keywords for this implementation:

`SPACE`	meaning a space.
`NL`	meaning a newline.
`TAB`	meaning a tab.
`SL`	meaning the imaginary startline character.
`SPACES`	meaning a sequence of one or more spaces.

XXX.7 S-variables

States number of S-variables and gives a table showing the usage of `S10` *upwards and their initial values. Most of the S-variables will already have been mentioned in previous Sections, particularly XXX.2.*

4 Testing

A set of test programs for checking an implementation of ML/I is available. These test programs are issued in the same format as other LOWL material. They are accompanied by sets of sample output from the test programs, as run on an existing implementation of ML/I. In general the output from the test programs should be easy to check at a glance since each line should be of form:

```
string1 SB string2
```

`SB` means “should be” and it is therefore an error if *string1* differs from *string2*. A few of the test programs check error detection, and the error messages produced by these require more careful checking.

Line numbers in messages may differ from those in the supplied sample results, due to additional macro calls used (for example) to select input or output streams before invoking the actual test. It may be possible to set `S2` to correct for this, and this will certainly make checking easier if it can be done.

4.1 Sequence of tests

Two of the tests programs, NAMES and ESCALATE, are designed for preliminary testing of a new implementation of ML/I. A common fault is a bug in the mapping macros in the LOWL extensions for defining table items, particularly the `HASH` and `THASH` statements. The NAMES test, which should be the first test fed to a new implementation, is designed to show up such faults. It does this by generating an error message which should include all the built-in macros (except `MCNODEF`, `MCNOWARN`, `MCNOINS` and `MCNOSKIP`, which are tested separately) and their delimiters.

After the NAMES test is working, the ESCALATE test should be applied. This, as its name implies, tests constructions of ever increasing complexity. It covers most of the features of ML/I, and hence when an implementation has successfully run it, that implementation can be said to be in reasonable shape and ready to face the buffeting that the other ML/I tests should give it.

The remaining tests are called ALTER, ERRORS, MACROTIME, OVERFLOW, OVERRIDE, S, SKIPINS and STRUCTURES. These tests should provide a comprehensive check, though no test programs can, of course, be completely foolproof. The names of the tests indicate the feature(s) they concentrate on.

These tests should be supplemented by a set of tests to check all the implementation-dependent features. In addition, these implementation-dependent tests should check the `TAB` layout keyword; this is not included in the main tests since not all implementations support tabs.

The next page contains a table which may be of use in monitoring the tests of the various versions of an implementation.

4.2 Sheet for checking test runs

This is an example of a test schedule sheet, suitable for recording tests on an implementation of ML/I.

Test	Version name or date of implementation										
NAMES											
ESCALATE											
ALTER											
ERRORS											
MACROTIME											
OVERFLOW											
OVERRIDE											
S											
SKIPINS											
STRUCTURES											
Impl. dep. 1											
Impl. dep. 2											
Impl. dep. 3											

Each time ML/I is changed, the new version name/date is entered into a fresh column and a tick placed against those tests that have successfully run. All the tests except NAMES and ESCALATE (and ALTER, if an implementation does not support MCALTER) should be applied to every version.

Concept Index