ML/I
Underlying data structures and functions

January 1977, October 2018

Stephen Fickas

R.D. Eager

Table of Contents

ML/I — Underlying data structures and functions

Preface

This document was originally written by Stephen Fickas. It 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. Some minor corrections and clarifications have also been made. The rewrite was carried out by Bob Eager.

The changes have been fairly minor, mainly to fit the document within the Texinfo conventions. Appendix C was added, documenting the subroutines and code sections in the L source. A contents list and index were also added. Reference is also made to ML/I character variables, which are not present in the L version, but are available in the C implementation.

1 Introduction

These notes are a by-product of my efforts in providing in-line documentation to the L version of ML/I. The following sections are arranged in the most conducive way for understanding the documented L code. The notes are general in nature, and for specifics the in-line documentation should be consulted.

1.1 Reference material

Some of the following material can be found on the ML/I website, at `http://www.ml1.org.uk`.

Three manuals/notes/listings should be obtained before attempting hand simulations of the documented L code:

1. *ML/I User's Manual, Fourth Edition* (or later), P.J. Brown, University of Kent at Canterbury.
2. *Implementing Software Using The L Language*, P.J. Brown, University of Kent at Canterbury.
3. The actual documented L code itself, P.J. Brown, University of Kent at Canterbury.

If only the LOWL or undocumented L version of ML/I are available, then also obtain *How ML/I Works*, P. J. Brown, 1973. For readers that are more interested in an L mapping, the following material will be useful:

a. *ANSI FORTRAN listing of ML/I*, Steve Fickas, Code 5200 Naval Electronics Lab, San Diego, CA 92152.
b. *ML/I Mapping Notes - FORTRAN*, Steve Fickas, Code 5200 Naval Electronics Lab, San Diego, CA 92152.
c. *The PL/I L-Map*, P. J. Brown, University of Kent at Canterbury.

The above material is available to anyone who agrees that it will not be used in any type of commercial venture.

1.2 Strategy

There is no belying the fact that the L code presents a complex system to be fully understood only after a number of simulated cases are tried. I have found the following procedure (not algorithm, for it does not necessarily halt) to be a good one in understanding the L code (• signifies newline):

1. After reading Chapter 4 [The Construction], page 9, sketch out a matched skip construction, using, say `<` as a primary name and `>` as a secondary and closing delimiter.
2. Call `SIMULATE(X<ABC>Y)`
3. Call `SIMULATE(X<A<BC>>Y)`
4. Using different `M`, `D` and `T` options, repeat steps 1, 2 and 3 until the skip is fully understood.
5. Sketch out the construction resulting from `MCDEF X AS Y`
6. Call `SIMULATE(A X B)`
7. Sketch out the construction obtained from `MCGO` in `MACNAMES` in the L data section.

8. Call `SIMULATE(MCGO L1•)`
9. Sketch out the construction obtained from `MCDEF` in `MACNAMES` in the L data section.
10. Call `SIMULATE(MCDEF X Y AS %A1.•)`
11. Sketch out the construction obtained from `MCINS` in `MACNAMES` in the L data section.
12. Call `SIMULATE(X Q Y Z)`
13. At this point, most of the major concepts of the L code should be understood. Still, there are many points still to be covered. I leave it to the reader to work out cases of nodes, nested calls, global vs. local definitions and error routines.

The subroutine SIMULATE is defined as:

```
SUBROUTINE SIMULATE (INSOURCE)
Starting at MBEGIN, hand-simulate
the L code, obtaining source input
from the passed parameter INSOURCE.
END SIMULATE
```

2 Low Level Data Structures

The easiest way to understand the L implementation of ML/I is first to understand the underlying data structures. There are three basic data areas in ML/I:

a. a large contiguous portion of memory used for stacks.

b. a contiguous portion of memory used for a hash table.

c. memory allocated to various pointer, number, switch and character variables. Note that the L version does not have character variables.

The size of each area is at the individual implementer's discretion. Each will be described below in more detail.

2.1 The stacks

See also Chapter 5 [FSTACK], page 15 and Chapter 6 [BSTACK], page 16.

ML/I makes use of two stacks called `FSTACK` (forwards stack) and `BSTACK` (backwards stack). Given a contiguous section of memory between `M` and `N` (`M<N`), `FSTACK` will grow towards `N` and have its bottom at `M`. Likewise, `BSTACK` will grow towards `M` and have its bottom at `N`. Currently, if they meet in the middle, ML/I is aborted.

2.2 The hash table

See [2], section 6.2.3.

The hash table is made up of a number of pointers (`LHV` being the number), four additional pointers and, finally, a switch variable. There is also an additional *bottom neighbour* pointer.

a. The first `LHV` pointers.
It is up to each implementer to define a routine called `MDFIND`, which takes the current atom being scanned and produces a number `K` where `0 < K < LHV`, the value of `LHV` also being up to the implementer. It is hoped that `MDFIND` will efficiently scatter `K` within this range. As various macros, inserts, etc. are defined, they will be added to the appropriate chain headed by one of these pointers.

b. The four additional pointers
These pointers serve a complex purpose and will probably not be fully understood until moving through the code. Briefly, they are used in conjunction with routine `CKVALY`, to turn off any local definitions when a `MCNODEF` is called, any local skips when a `MCNOSKIP` is called, etc. (see [1], section 5.2.5). The initial value for all four is any number greater than a possible pointer value.

c. The global warning switch
Used in conjunction with a call on `MCWARNG`. Initially `7`, denoting no warning active, changed to 6 when `MCWARNG` is called.

d. Bottom Neighbour
Although not strictly part of the hash table, a pointer is added to the end of the hash table every time one is stacked (i.e. each time a new level is entered). This pointer points to the start of the previously stacked hash table (or `NULLPT` for the earliest version), and in this way the stacked hash tables are chained together.

Note
Although it was initially stated that only one hash table exists, there now seems to be a proliferation of hash tables. Actually, storage need be allocated for only one hash table; in processing, ML/I will stack and unstack various copies of this original on `BSTACK`.

2.3 Variables

See [2], section 3.1.

Each variable type can use an implementer defined number of bits, words or bytes, depending on the mapping of the `OF` function.

3 Miscellaneous Data Types

3.1 Pointers

There are two types of pointers used in ML/I; relative and absolute. Relative pointers (actually, numbers) are used only within the stacks and are defined relative to their location. For example, if a block of 10 pointers are stacked on `FSTACK`, and we wish the last pointer to point relatively to the first, we would set it to -9.

Absolute pointers correspond with the usual notion of a pointer, their value being the address of some memory location.

```
+-----+
|  R  | flags a relative pointer
+-----+
|  A  | flags an absolute pointer
+-----+
```

3.2 Arrays of numbers

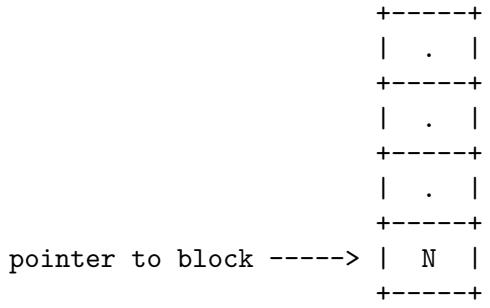
Most blocks of numbers (i.e. permanent variables, system variables, etc.) have the following format:

```
                          +-----+
                          |  .  |
                          +-----+
                          |  .  |
                          |  .  | values
                          |  .  |
                          +-----+
                          |  .  |
                          +-----+
pointer to block -----> |  N  | number of above values
                          +-----+
```

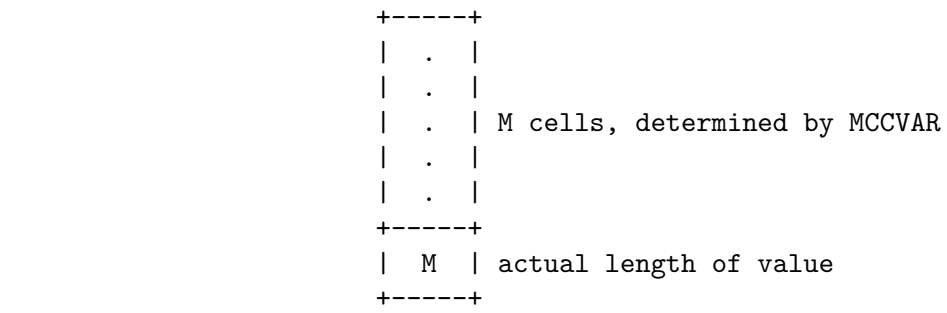
where the values are stored in reverse order. For example, `P0` would be the cell above `N` and `PN-1` would be the top cell.

3.3 Arrays of characters

If character variables are implemented, they are stored as a fixed number of character cells preceded by the actual number of cells used. The fixed number is determined by the second argument to the first call of `MCCVAR`, and is stored in `CVSIZE`. The variables are stored in the following format:

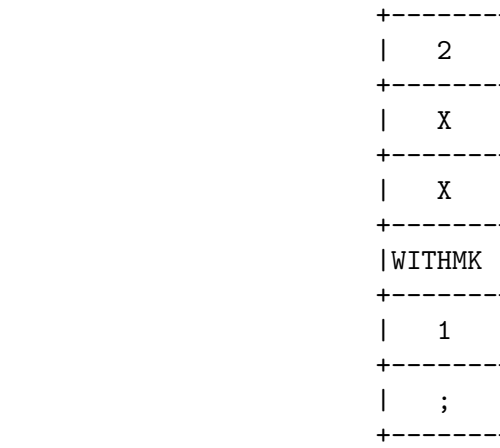


where the values are stored in reverse order. Each value looks like this:



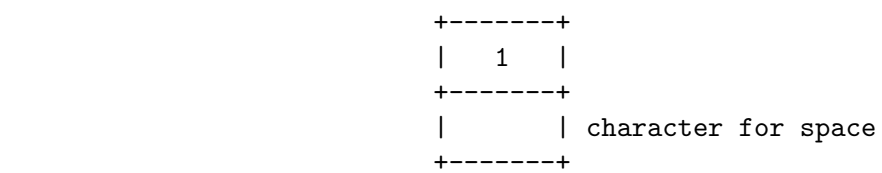
3.4 LIDs

A `LID` is a number `N` followed by `N` characters. One or more `LID`s can occur in sequence. If there is more than one, they are separated by either a `WITHMK` or a `WTHSMK`. As an example, `XX WITH ;` would be represented as

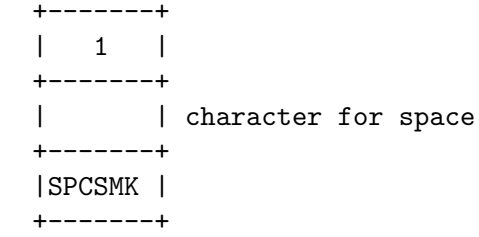


Following are the definitions for various keywords:

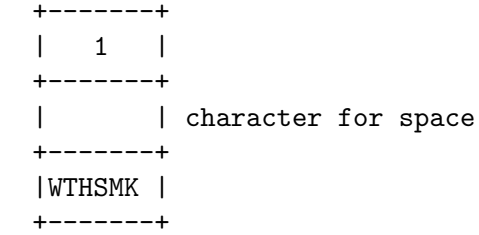
a. `SPACE`



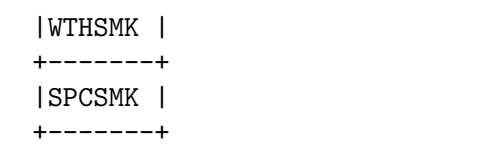
b. SPACES



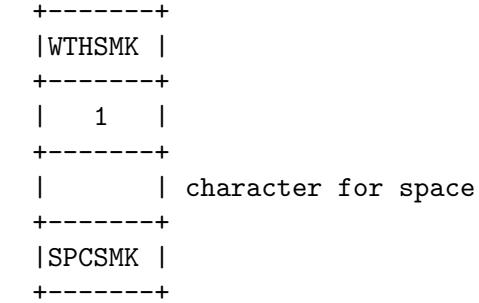
c. SPACES WITH, SPACES WITHS, SPACE WITHS



d. WITH SPACES, WITHS SPACE



e. WITHS SPACES



3.5 The ERBLOC

There is a contiguous block of storage required for the error block (ERBLOC), which is used by the error routines. This can be reserved on the top of FSTACK, or else in variable storage.

4 The Construction

Perhaps the quintessential structure of ML/I is the construction. Each time a `MCDEF(G)`, `MCWARN(G)`, `MCINS(G)` or `MCSKIP(G)` is processed by ML/I, a new construction is built. Constructions exist also for each ML/I operation macro (e.g. `MCDEF`), but not necessarily on the stack. All local and global constructions reside entirely on one of the two stacks. A construction or structure representation is as shown in the following sections.

4.1 Primary construction

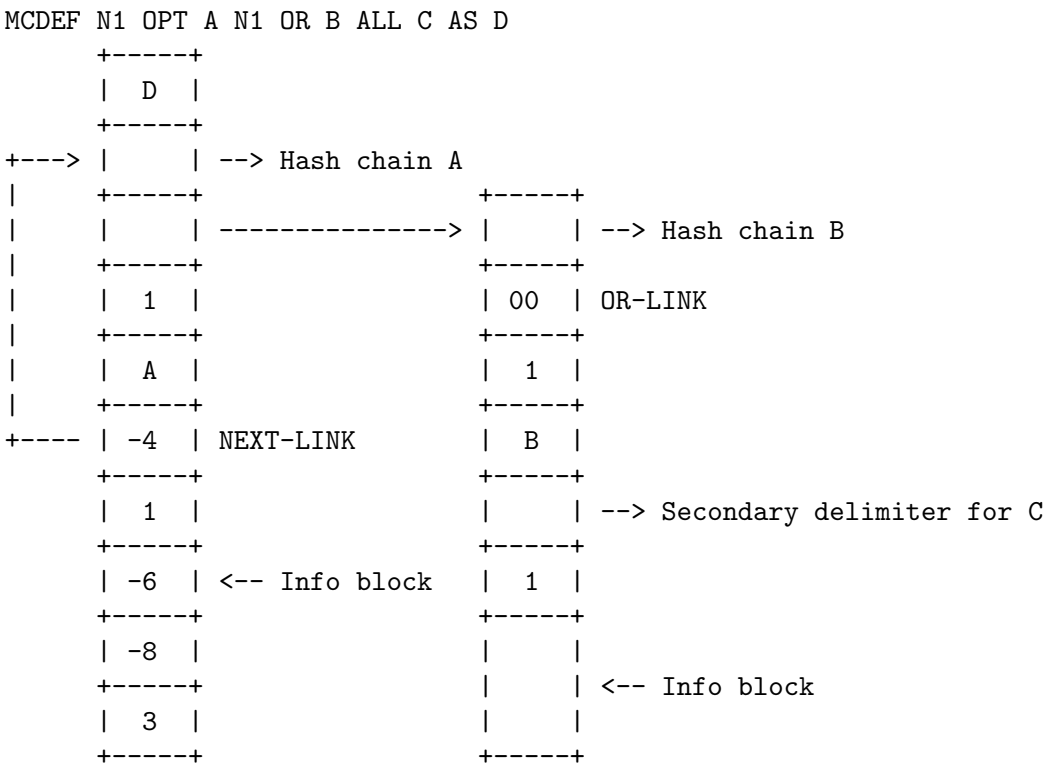
```
+-----+
|  .  |
|  .  | Replacement text (see type 1 below)
|  .  |
+-----+
|  A  | Hash chain to next construction
+-----+
|  R  | OR-LINK to alternative name of same construction
+-----+
|  .  | One or more LIDs representing the name of the
|  .  | primary construction (more than one in the
|  .  | case of WITH)
+-----+
|  R  | NEXT-LINK to next delimiter
+-----+
|     | Construction type (0 through 4)
+-----+
|  .  |
|  .  | Information block (depends on type)
|  .  |
+-----+
```

The items above are, in more detail:

a. Hash chain
This is used to form a list which originates in the hash table. Each primary construction on the list has been mapped into the same `K` and thus belongs to an equivalence class defined by `MDFIND`.

b. `OR-LINK`
Normally `ENDCHN`, else used to link together alternative names (i.e. primary constructions) for this construction. Before studying the following example, it will probably be necessary to review [1], section 5.1.3.

An example macro definition, and its corresponding representation, is:



ML/I will then recognise as a macro call:

```
    B ..... C
  A B ..... C
A A B ..... C
```

etc.

c. NEXT-LINK
Points at the next delimiter to be matched.

This is normally a secondary delimiter, but in some cases may be another primary construction (see (b) above).

d. Construction type

```
0 for STOP-MARKER
1 for MACRO
2 for WARNING-MARKER
3 for INSERT
4 for SKIP
```

4.2 Information blocks

a. Type = 0 STOP-MARKER
No information block

b. Type = 1 MACRO
There are two types of macros; user defined substitution macros, and ML/I defined operation macros. They can be differentiated by the first element of their information

block; if this is < 0 or 2, this is a substitution macro; if it is 0 or 1, this is an operation macro. Other values are unused and undefined.

A substitution macro information block looks like this:

```
+-----+
|  2  | optional STRMK
+-----+
|  R  | --> negative offset to end + 1
+-----+
|  R  | --> negative offset to start
+-----+
|     | number of temporary variables
+-----+
```

If the optional `STRMK` is present, then this is a straight scan macro (see [1], section 2.9).

The two negative offsets refer to the replacement text in the primary construction.

As an example, after ML/I had fully processed

```
MCDEF X AS Y
```

the following construction would appear on the hash chain associated with `X`:

```
+-----+
|  Y  | replacement text
+-----+
|     | --> hash chain X
+-----+
| 00  | OR-LINK
+-----+
|  1  |
+-----+
|  X  |
+-----+
| 00  | NEXT-LINK
+-----+
|  1  |
+-----+
| -6  | Info block
+-----+
| -8  |
+-----+
|  3  |
+-----+
```

An operation macro information block looks like this:

```
+-----+
|  .  | LOCMK (1) for local macros, else OPMK (2)
+-----+
|  .  | Operation type (0-15)
+-----+
```

c. Type = 2 WARNING-MARKER

No information block

d. Type = 3 INSERT

```
+-----+
|  .  | PINSMK (2) or UINSMK (3)
+-----+
```

`PINSMK` for protected insert, `UINSMK` for unprotected insert (see [1], section 2.6.8).

e. Type = 4 SKIP

```
+-----+
|  .  | flags (only least significant three bits used)
+-----+
```

Bit 0 = 1 if matched skip (`M` option) (least significant bit)
Bit 1 = 1 if text copy (`T` option)
Bit 2 = 1 if delimiter copy (`D` option)

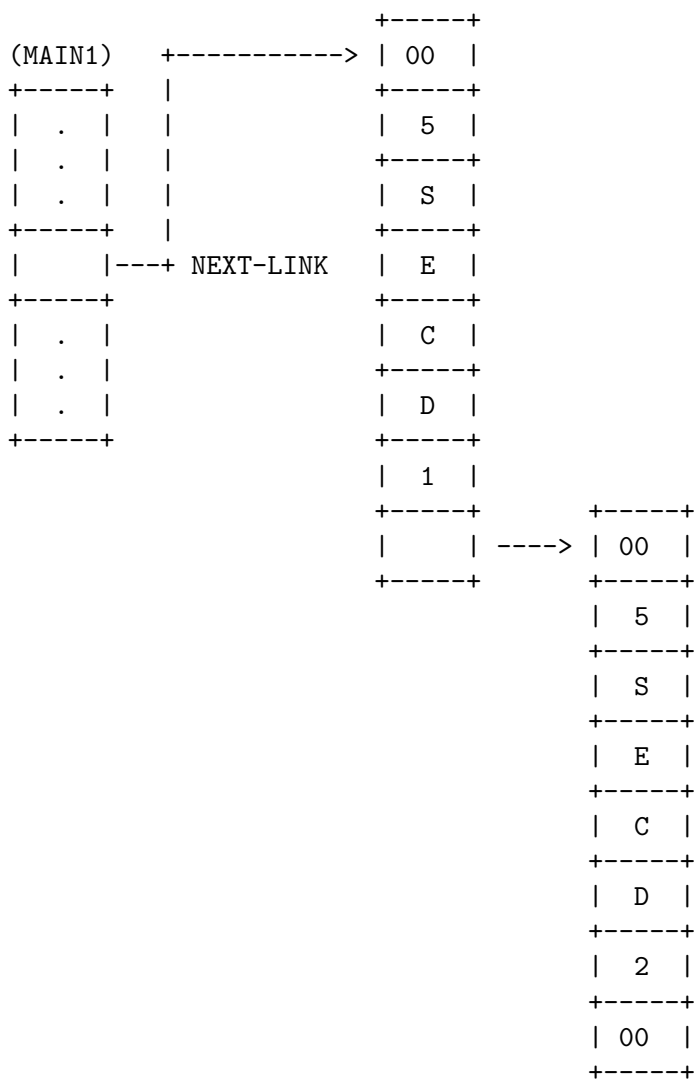
4.3 Secondary delimiters

```
+-----+
|  R  | OR-LINK
+-----+
|  .  |
|  .  | LID(s)
|  .  |
+-----+
|  R  | NEXT-LINK
+-----+
```

As can be seen, secondary delimiters have a similar but abbreviated format to the primary construction. They are used to hold delimiters which follow the primary name. As an example,

```
MCDEF MAIN1 SECDl SECD2 AS ...
```

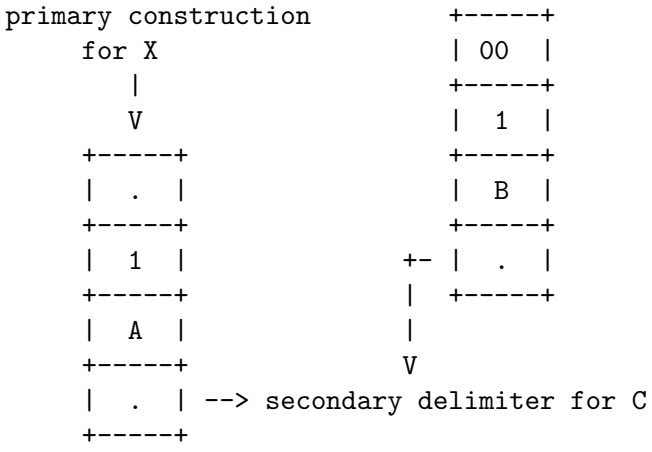
would result in



The OR-LINK is used to chain together delimiters found in an option block. As an example:

```
MCDEF X OPT A OR B ALL C AS ...
```

would result in:



The NEXT-LINK can be:

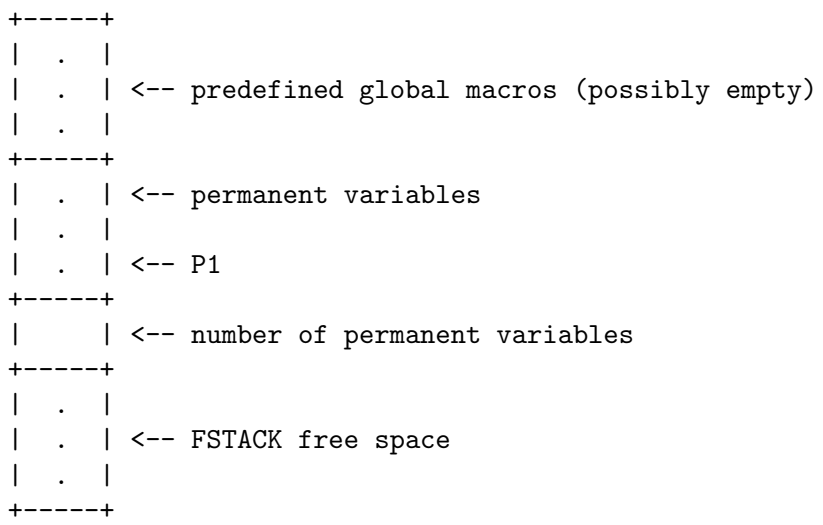
a. a pointer to the next delimiter required;

b. `ENDCHN`, signifying this is the last delimiter, or

c. `EXCLMK`, signifying this is the last delimiter, and in addition that this is an exclusive delimiter (see [1], section 3.5).

5 FSTACK

The top pointer of FSTACK is FFPT, which points at the first available cell (contrast with LFPT). After finishing the initialisation code at MBEGIN, FSTACK will be as follows:

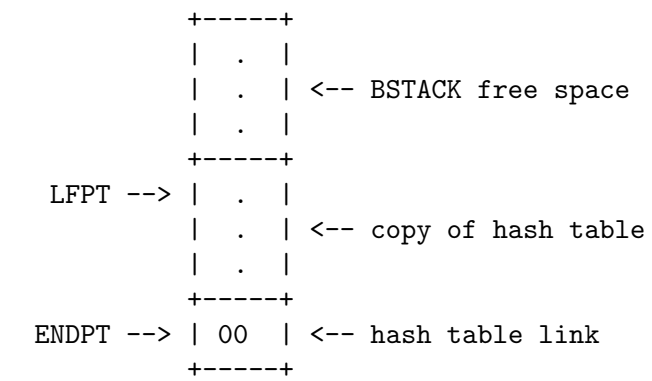


All global constructions are retained on the bottom of FSTACK. Since the ML/I macros defined in the MACNAMES section are global, they may be placed here. When the user calls upon ML/I macros such as MCDEFG or MCSKIPG, they are defining additional global constructs which must be included on FSTACK. This is accomplished by inserting the new global construction in between the current global constructions and the top of the permanent variables (see MKROOM routine), moving everything down to make room.

If character variables are implemented, a single value of zero is stacked on FSTACK immediately after the number of permanent variables; this signifies the number of character variables (initially none), and CVARPT is set to point to this.

6 BSTACK

The top pointer of BSTACK is LFPT, which points at the last used cell. After finishing the initialisation code at MBEGIN, BSTACK looks as follows:



All local constructions are eventually placed on BSTACK. Note that this means a hash chain pointer to a global definition will always be less than LFPT. This fact is used in differentiating between the two.

7 Recognition

Once the basic data structures of ML/I are understood, it is left to study the various ways ML/I manipulates these structures to produce the desired results. There are two basic actions which ML/I performs:

a. construction evaluation, which involves building new constructions, setting macro time variables, etc. and

b. recognising previously defined constructions.

Since the second is a prerequisite to the first, it may be desirable to understand this section before moving on to the next, which deals with the evaluation of a recognised construction. Ignoring the STOP-MARKER, there are four types of construction:

a. MACRO

b. WARNING-MARKER

c. INSERT

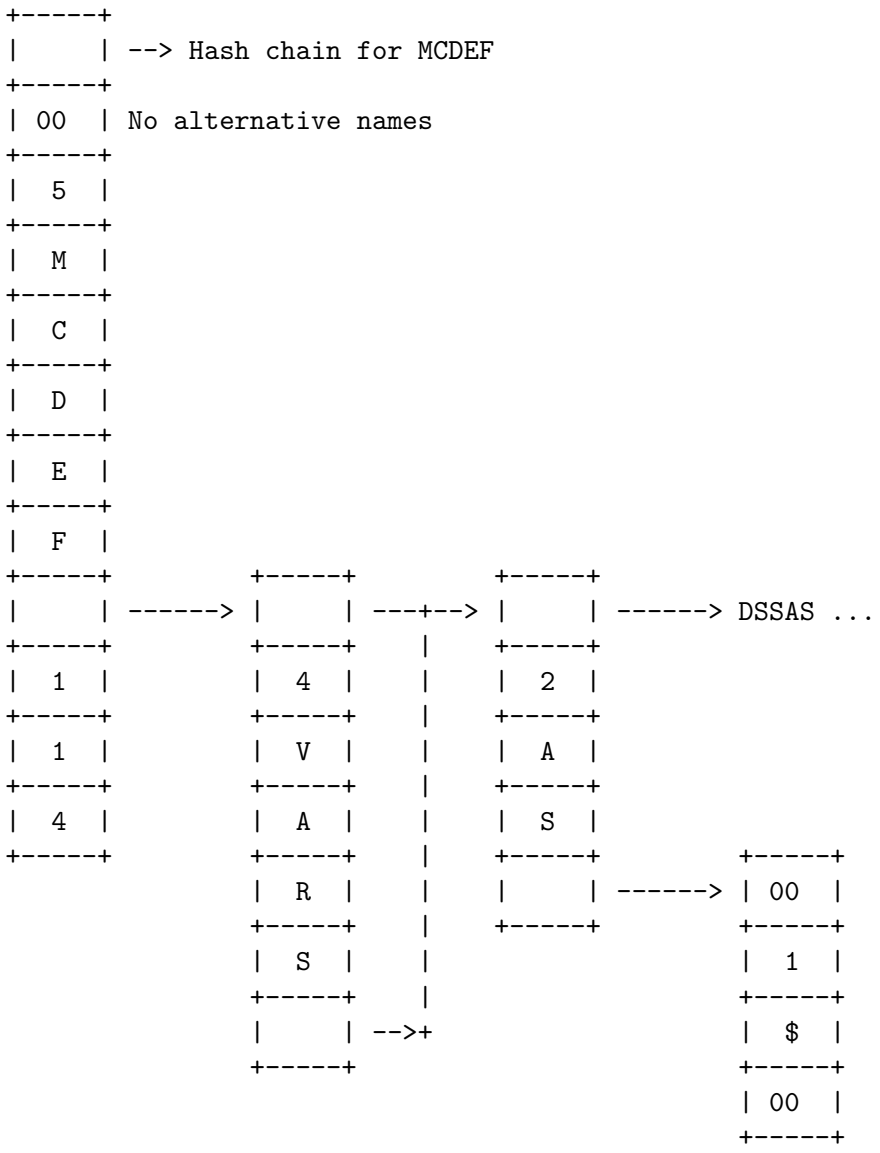
d. SKIP

All are evaluated in similar fashion, but the MACRO construction has been chosen to follow as an example, mainly because it is the most difficult of the four. Let us assume that the user has typed in:

```
MCDEF A B C AS D
```

7.1 Recognising the primary name

The section of code at `BSNAME` is responsible for searching the primary constructions, looking for a match on the current atom. In our example, the current atom is `MCDEF`, and a match will be found with the primary construction set up in the `MACNAMES` section. Following is a diagram of the `MCDEF` construction set up in the `MACNAMES` and `DELS` sections. The `$` sign is used to represent a newline. Readers may wish to review these two sections for their own verification.



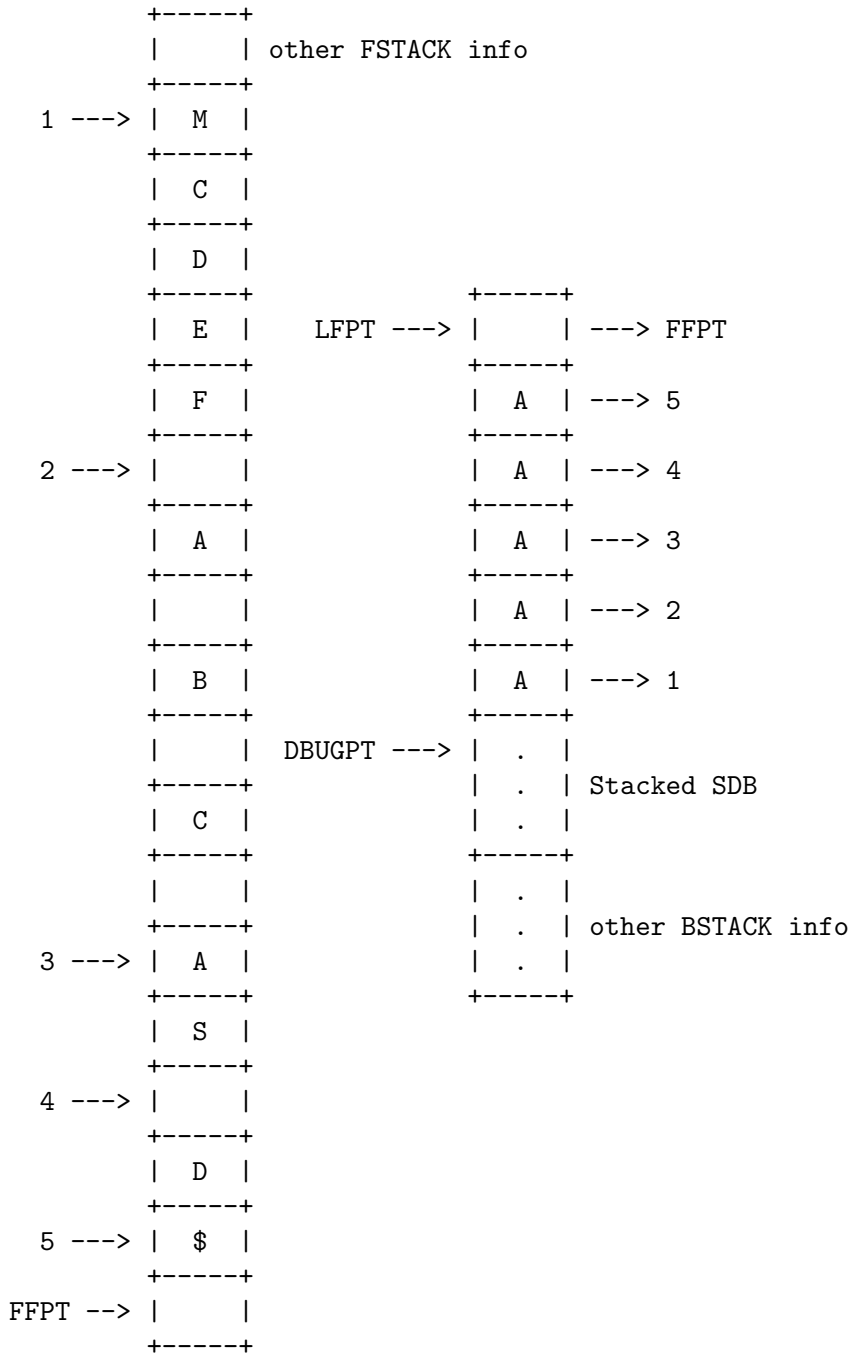
The TEBEST routine will decipher this as a MACRO type construction, and set the variable BESTPL so as to jump at BSSWIT accordingly (i.e. to CALLO).

7.2 Delimiter recognition

After the primary name has been matched, ML/I attempts to find each of the secondary delimiters defined by the NEXT-LINK chain. In our example, ML/I will continue reading in characters (i.e. the first argument A B C) until either a VARS, AS, or SSAS is matched. When the AS of our example is read, CMPARE will match it and TEBEST, after determining that it is a secondary delimiter, will set BESTPL so as to jump to DELLO at BSSWIT.

ML/I will determine that AS is not the last or closing delimiter, and so continue reading characters (i.e. the second argument D) until the next delimiter (newline) is matched, again shifting control to DELLO. ML/I notes that this is the last delimiter, so searching is halted. One by-product of the above actions is an argument vector, which is constructed on BSTACK.

Below is the status of the two stacks at this point; once again, the $ sign is used to represent a newline.



7.3 Nested constructions

If, while searching for the delimiters by the procedure described above, a nested construction call is found, then note the following actions:

a. Nested macro call

In our example, suppose that `MCDEF A C AS Z` had been previously typed. While searching for the `AS` in our current example, ML/I would have discovered the nested call on `A`. The action of ML/I in this case would then be:

1. suspend the current search for `AS`,
2. process the macro `A` in the same fashion described earlier in this section, i.e. find all delimiters, and
3. resume looking for `AS`.

Note that at this stage `A B C` is not replaced by `Z`, but is merely scanned over so that the search for `AS` can be resumed.

b. Nested skip call
Similar to above in that the current search is suspended until a closing skip delimiter is found, whereupon the suspended search is resumed.

c. Nested insert call
In a similar way to the previous two, while the current search is suspended, the insert delimiters will be matched, after which the suspended search is resumed.

It should be clear from the above description why it is not allowed to use macros in place of delimiters in the hope that they will be evaluated and reduced to the desired delimiter. As an example, suppose we are given the following definitions:

```
MCDEF Q AS B
MCDEF A B AS . . .
```

then we might expect the call

```
A ... Q
```

to be correctly matched. We know that it will not, for even though `Q` will be recognised as a macro, it will not be evaluated (i.e. replaced by `B`), and so ML/I will vainly search for the closing delimiter `B`.

The next section will describe the actions of ML/I after construction evaluation.

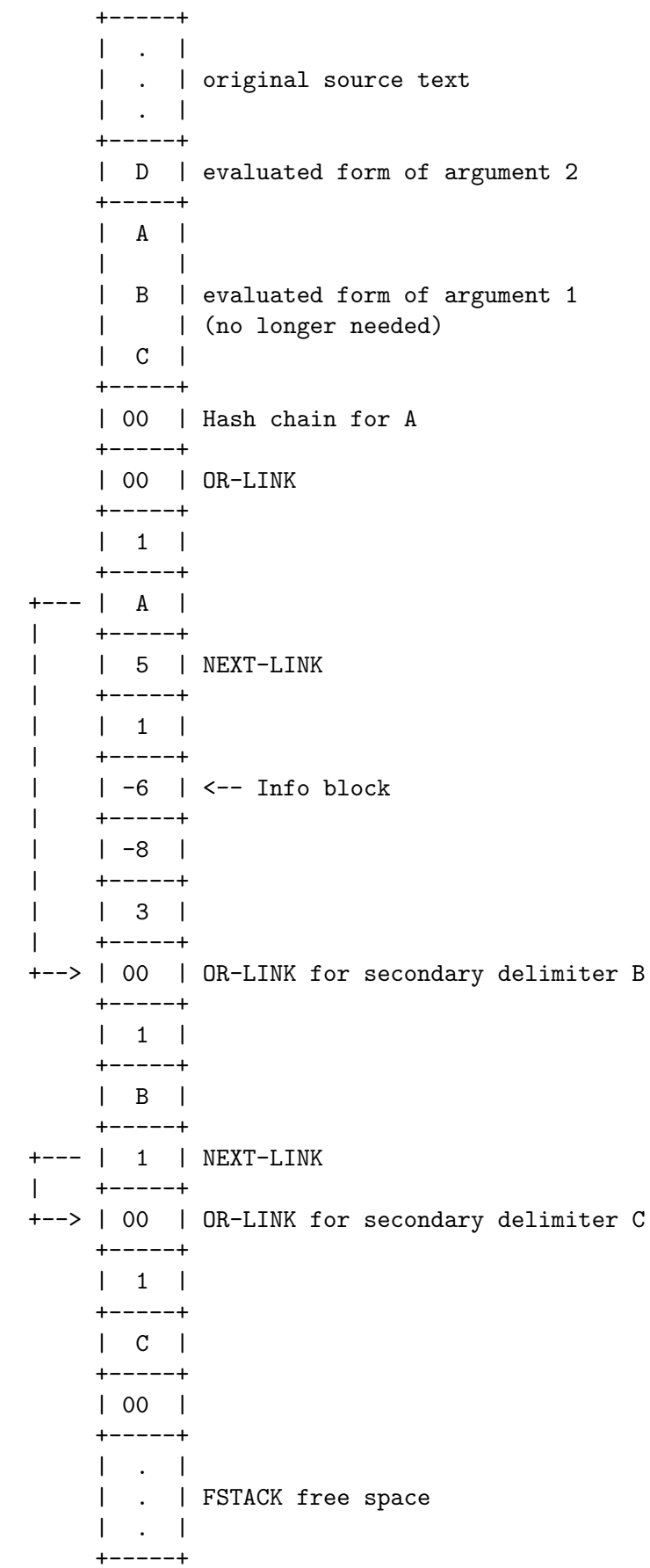
8 Construction Evaluation

This section deals with the actions taken by ML/I once a construction has been recognised. When an example is called for, we will carry on with the example started in Chapter 7 [Recognition], page 17, i.e. `MCDEF A B C AS D`.

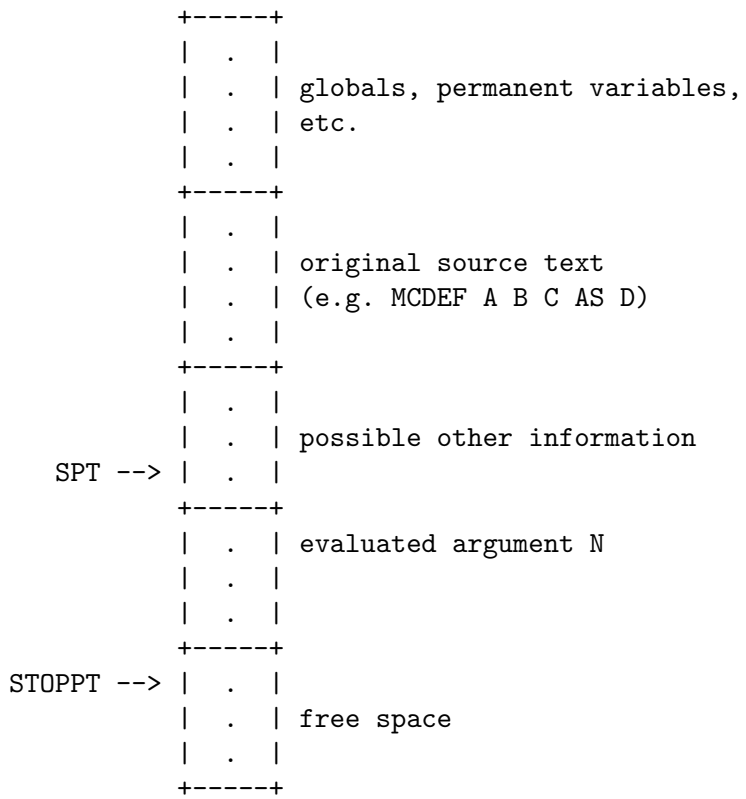
8.1 Evaluating arguments – the `STKARG` routine

In Chapter 7 [Recognition], page 17, ML/I skipped over any nested constructions found while searching for delimiters. Before evaluation can proceed on `MCDEF`, each argument must be evaluated to catch any nested calls. The `STKARG` routine is used by ML/I for this purpose, evaluating a specific argument. When `STKARG` is called, `ARGNO` contains the argument number to be evaluated. `STKARG` then finds where the argument starts and ends in `FSTACK`, and jumps back to the construction recognition loop to scan the argument. Basically, we are back to Chapter 7 [Recognition], page 17, trying to recognise a primary construction. We can see then that `STKARG` is a recursive routine, for if any constructions are recognised in an argument, they in turn will eventually lead to another call on `STKARG`, potentially *ad infinitum*.

The following diagram shows the call `MCDEF A B C AS D` before link-up.


original source text
D
evaluated form of argument 2
A
B
evaluated form of argument 1
(no longer needed)
C
00
Hash chain for A
00
OR-LINK
1
A
5
NEXT-LINK
1
-6
<-- Info block
-8
3
00
OR-LINK for secondary delimiter B
1
B
1
NEXT-LINK
00
OR-LINK for secondary delimiter C
1
C
00
FSTACK free space

When control does return from STKARG (LINK-BACK) for the final time, we are guaranteed that all nested calls have been evaluated, and FSTACK will be as follows:



8.2 Building a construction

Depending on the construction type (i.e. MCSET, MCDEF, user macro, etc.), prescribed actions are carried out on the evaluated arguments. To show what these actions are in the MCDEF case, we will carry on with the example started in Chapter 7 [Recognition], page 17. After finding the closing delimiter, ML/I will transfer control to DEF, the MCDEF handler. There are two arguments to worry about: the replacement text D and the structure representation A B C. The strategy ML/I will use is:

a. evaluate the replacement text and place on FSTACK

b. evaluate the structure representation and place on FSTACK

c. build a construction as defined in Chapter 4 [The Construction], page 9 on FSTACK using results from a. and b.

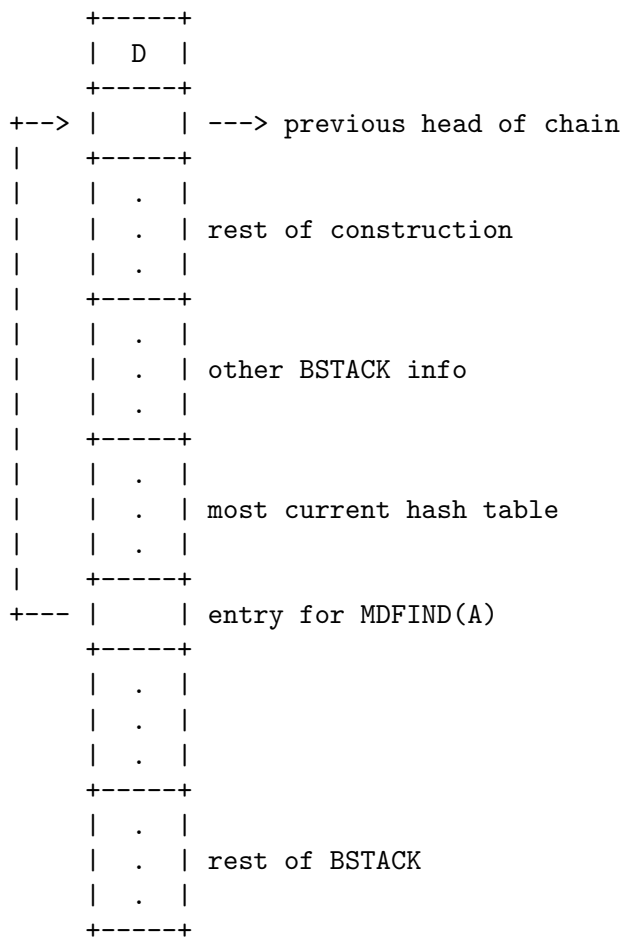
The result can be seen in Section 8.1 [Evaluating arguments – STKARG], page 21. If arguments 1 or 2 had contained any nested calls, their evaluated forms would have reflected the results of these calls. In our example, since no nested calls were involved, the call to STKARG was almost nothing more than a straight copy from the original source text to the evaluated form, (almost, because leading and trailing spaces are deleted).

In arriving at the diagram in Section 8.1 [Evaluating arguments – STKARG], page 21, we have glossed over a large amount of ML/I processing in the EVTREE section, and in the

GETDEL routine. These sections of code deal with the sticky details of parsing a structure, and will need to be worked through by hand to appreciate their function.

8.3 Link-up

The final step of ML/I is to link the newly built construction into the appropriate hash chain, in our case MDFIND(A). Preliminary to this link-up, ML/I will move the construction up adjacent to the replacement text, overwriting evaluated argument 1 which is no longer needed. Since our construction is a local one (MCDEF instead of MCDEFG), it will be moved to the top of BSTACK. Next, the correct hash chain will be found and ML/I will insert the new construction at the head of the chain. BSTACK will be as follows:



This concludes ML/I processing of MCDEF A B C AS D.

References

1. Brown, P.J. and Eager, R.D., *ML/I User's Manual*, Sixth Edition.
2. Brown, P.J. and Eager, R.D., *Implementing software using the L language*.

Appendix A Description of uses of variables

The following summarises the uses of important variables in the MI-logic of ML/I. Note that line numbers may vary slightly, depending on the exact version of the L code available.

Variable	Declared in line	Meaning
ALLPT	61	head of chain of nextlinks to be attached to delimiter following ALL
ARGCT	13	in SDB
ARGLEN	49	length of value of current argument
ARGNO	24	in SDB
ARGPT	19	in SDB
BESLIN	123	best-so-far value of LINECT
BESPT	96	best-so-far value of SPT
BESTPL	120	switch value used in basic scan routine
BFNDPT	87	best-so-far value of FNDPT
BINDIC	111	best-so-far value of INDIC
BINFPT	89	best-so-far value of INFOPT
CALTYP	124	first number in information block
CHANPT	94	used in CHAIN FROM operation
CHLINK	116	used in CHAIN FROM operation
CINFPT	103	points at information block in primary construction
CLLFPT	101	points to top entry after scanning information is stacked
CONSW	150	for syntax checking (see EVTREE and GETDEL)
COPDSW	131	for skips. Reflects setting of delimiter option.
COPTSW	130	for skips. Reflects setting of text option.
CVARPT	72	points at character variables
CVNUM	74	number of character variables
DBUGPT	20	in SDB
DBUGSW	25	in SDB
DELCT	145	count of delimiters
DELPT	104	head of chain of delimiters being searched for
ENDPT	71	points at end of BSTACK
ERIAPT	95	points at value of operation macro argument
ETEMPT	84	points at error block (temporary storage for EDB)
EXIDPT	140	temporary storage for IDPT
EXITSW	151	for syntax checking (see EVTREE)
FFPT	90	points to first free location on FSTACK
FLAGPT	158	points at flag
FNDPT	86	points at LID when a name is found
GHSHPT	845	points at global hash table
GLBWSW	705	value 6 if there is a global warning; value 7 otherwise
HASHPT	30	in SDB
HTABPT	97	points at current hash table
IDLEN	119	length of current identifier

IDPT	98	points at current identifier
INDIC	115	contents of nextlink
INFFPT	34	in SDB
INFOPT	88	points beyond currently matched LID
INSW	132	to set DBUGSW, and as implied parameter to SETPTS
INVOCT	113	count of macro calls
KEYSW	149	for syntax checking (see EVTREE and GETDEL)
KNPT	83	temporary storage
LABPT	33	in SDB
LEVEL	112	level of macros and inserts
LFPT	73	points at top of BSTACK (last used location)
LINECT	23	in SDB
LINKPT	48	link for STKARG
LNODPT	139	points at topmost node entry on BSTACK
MASKSW	129	mask used to indicate which types of construction are recognised
MCHLIN	22	in SDB
MEVAL	122	miscellaneous. Used for numerical values calculated at macro time
MHSHPT	46	value of HASHPT when macro was called
MTCHPT	31	in SDB
MTYPE	175	type of items to be printed
NARGPT	102	points at argument vector + 1
NDEFPT	143	destination of newly defined construction
NEGVAL	162	indicates if result is to be negative
NESTLV	109	nesting level of calls and skips during scanning
NNODPT	138	points at current node entry on BSTACK
NODEPT	137	head of chain of links to be attached to next delimiter
NODESW	148	for syntax checking (EVTREE and GETDEL)
NTYPSW	53	type of construction being defined
OFFSET	121	offset in hash table
OHSW	37	in SDB
OLDSPT	142	previous value of SPT
OLIDPT	157	temporary storage for IDPT
OLLFPT	141	previous value of LFPT
OP1	160	LHS operand
OPLEV	110	level of operation macros and inserts
OPSW	164	"operation expected" switch (see GETEXP)
OPTHPT	663	head of orlink chain
OPTLEV	146	level of OPT-ALL brackets
OPTPT	62	last entry on orlink chain
OPTYP	50	type, e.g. global, local, insert. Also other uses
PARNM	117	formal parameter of type number
PARPT	91	formal parameter of type pointer
PARSW	128	formal parameter of type switch
PRSTPT	172	start of current hash table (see PRCTXT)

PRTAPT	173	counts down hash table (see PRCTXT)
PRTBPT	174	follows down hash chains (see PRCTXT)
PVARPT	72	points at permanent variables
PVNUM	74	number of permanent variables
SKIPLV	108	level of skip nesting
SKLIN	36	in SDB
SKVAL	35	in SDB
SPT	21	in SDB
SQNUM	51	safe variable, miscellaneous uses
SQPT	47	safe variable, miscellaneous uses
SQSW	52	safe variable, miscellaneous uses
STAKPT	18	in SDB
STFFPT	100	points at start of global macros
STOPPT	32	in SDB
SUM	161	running total
TEMAPT	93	temporary pointer
TEMP	114	temporary number
TEMPSW	127	temporary switch
TEMAPT	92	temporary pointer
TEMPT	92	temporary pointer
TIDPT	99	temporary storage for IDPT
TLINCT	125	temporary storage for LINECT
TOPSPT	45	points at latest OPDB on BSTACK
TSPT	100	temporary storage for SPT
TVARPT	29	in SDB
TYPE	118	type of construction name
VARPT	156	points at vector of variables
VARSW	165	"variable expected" switch (see GETEXP)
WNIDPT	106	temporary storage for IDPT
WNSPT	105	temporary storage for SPT
WSW	166	written argument or delimiter

Appendix B Uses of variables in SDB

This Appendix describes the uses of the variables declared in the Scanning Description Block (SDB), and their values in each of the main scanning states.

Name of variable	1. Scanning source text	2. Scanning replacement text	3. Scanning argument or delimiter	4. Evaluating operation macro or first insert
ARGCT	counts number of arguments when nested construction is scanned			not used
STAKPT	NULLPT	points at latest SDB on stack		
ARGPT	NULLPT	points at argument vector	containing text value	calling value
DBUGPT	not used	points at orlink preceding macro name	points at argument vector in which argument or delimiter is included	points at argument vector
MCHLIN	set to current value of LINECT when a nested construction is encountered			not used
LINECT	line count of current text			not used
ARGNO	not used	not used	number of argument or delimiter	number of argument currently being processed
DBUGSW	0	1	2 for operation macro argument 4 for substitution macro argument 6 for delimiter	5
HASHPT	points at local hash table		U insert: calling value P insert: containing text value	calling value
TVARPT	NULLPT	points at temporary variables	containing text value	calling value
MTCHPT	when a nested construction is encountered, this is set to point at the orlink of this construction			used as workspace

SPT	points at last scanned character			used in scanning values of arguments
STOPPT	NULLPT	points one beyond last character of current text		points one beyond end of latest argument
LABPT	NULLPT	head of chain of labels		not used
INFFPT	points at start of source text on FSTACK	not used	for operation macro argument: 4; otherwise undefined	value of FFPT when operation macro was called
SKVAL	if not scanning call or insert, then number of designated label for forward MCGO, zero if none in progress. If scanning call or insert, then $(-1 -$ (above value))			not used
OHSW	true if a new hash table has been stacked for this level; false otherwise			not used
SKLIN	if in forward MCGO, then line number in which MCGO occurred			not used

Appendix C List of subroutines and code sections in L

The following lists all of the sections, subroutines and other important areas of code in the L source.

Name	Section	Type	Declared in line
ADVNCE	MAINSUBS	Subroutine	383
BUMPFF	MAINSUBS	Subroutine	395
CHATOM	MAINSUBS	Subroutine	406
CHEKID	MAINSUBS	Subroutine	418
CKVALY	MAINSUBS	Subroutine	430
CMPARE	MAINSUBS	Subroutine	442
CORECT	MAINSUBS	Subroutine	460
DECALV	MAINSUBS	Subroutine	471
DECLF	MAINSUBS	Subroutine	484
DEFSUBS		Section	1448
ENCALL	MAINSUBS	Subroutine	495
ENVPR		Section	2116
ER1TST	MAINSUBS	Subroutine	511
ERMTST	ERR	Subroutine	1773
ERR		Section	1712
ERSIC	ERR	Subroutine	1743
ERSNW	ERR	Subroutine	1759
ERTEST	DEFSUBS	Subroutine	1450
GARGCH	MAINSUBS	Subroutine	523
GETDEL	DEFSUBS	Subroutine	1460
GETEXP	MAINSUBS	Subroutine	535
GMEADD	MAINSUBS	Subroutine	584
GSATOM	MAINSUBS	Subroutine	610
GSRATM	DEFSUBS	Subroutine	1610
GTATOM	MAINSUBS	Subroutine	626
INVALS		Section	8
JOINCH	DEFSUBS	Subroutine	1622
KEYSRC	DEFSUBS	Subroutine	1639
LUDEL	MAINSUBS	Subroutine	652
LULAYK	MAINSUBS	Subroutine	665
MAIN		Section	26
MAINSUBS		Section	380
MCALTERMAC	OPMACS	Code for	1024
MCDEFMAC	OPMACS	Code for	1238
MCGOMAC	OPMACS	Code for	1056
MCINSMAC	OPMACS	Code for	1260
MCLENGMAC	OPMACS	Code for	1135
MCNO---	OPMACS	Code for	1216
MCNOTEMAC	OPMACS	Code for	1148
MCPVARMAC	OPMACS	Code for	1161

Index

S

T

U

W