University of Wollongong
Research Online

Department of Computing Science Working Paper Series

Faculty of Engineering and Information Sciences

1978

SLMP: Source Library Maintenance Package implementation notes

Ian Almond
University of Wollongong, uow@almond.edu.au

Recommended Citation
Almond, Ian, SLMP: Source Library Maintenance Package implementation notes, Department of Computing Science, University of Wollongong, Working Paper 78-9, 1978, 15p.
http://ro.uow.edu.au/compsciwp/6

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

SLMP

Source Library Maintenance Package

Implementation Notes

Ian Almond

University of Wollongong

ABSTRACT

SLMP is a Source Library Maintenance Package written to enable a group of 'source decks' or programs to be stored on a common library and provide a system for maintaining a record of all modifications made to each program. It is designed to be used as a UNIX command which will accept from the standard input a set of commands and 'update directives' which control the manipulation of decks on the library and line oriented modifications to specific decks.

December 19, 1978

SLMP

Source Library Maintenance Package

Implementation Notes

Ian Almond

University of Wollongong

TABLE OF CONTENTS

1. INTRODUCTION 2

2. DEVELOPMENT 2

3. IMPLEMENTATION 5
3.1. File Format 5
3.2. I/O within SLMP 7
3.3. Brief Descirption 8
3.4. Detailed Descripton 9
3.4.1. ^add 9
3.4.2. ^delete 9
3.4.3. ^replace 9
3.4.4. ^list 9
3.4.4.1. getmods 11
3.4.4.2. moddeck 12
3.4.4.3. ldeck 13
3.4.5. ^extract 13
3.4.6. ^update 13

4. FUTURE DEVELOPMENTS 13

5. BUGS 13

6. CONCLUSION 14

December 19, 1978

1. INTRODUCTION

This document has been produced in conjunction with the Source Library Maintenance Package (SLMP) developed on the Interdata 7/32 operating at the University of Wollongong. The package was developed due to the lack of an adequate system for maintaining source code. A need was expressed for a system that would provide a method of storing and updating source code on a library in such a way that all modifications that are made are identified, thus allowing the original code to be obtained at any time. An extension to this thought was to identify each update with a string of characters supplied by the user and provide the facility for obtaining a copy of the code as it was at any point in time.

The descriptions of the data structures used in the 'C' code given in the apropriate sections of this document provide a general overview of the design of the package. Almost all changes to the design and unimplemented features discussed will have a direct affect on one or more of the data structures.

A call to SLMP is made with the name of the library and a mode which indicates whether the library is a new one or already exists and the type of commands to be used. The commands are read from the standard input and instruct the package to perform a certain function on a specified deck. Following three of the commands an optional set of 'update directives' can be supplied to perform modifications to the associated deck by deleting and inserting entire lines. The run is terminated by an end-of-file or the ^stop command.

2. DEVELOPMENT

During the course of developing the package, as problems arose and were overcome, and new features were added, the design of the system was continually being rethought and improved. The final design has resulted in a very useful and flexible system with plenty of room for the development of a wider and more sophisticated range of functions.

It was decided for the sake of simplicity to keep each source code deck on the library as a contiguous block of data and maintain a directory describing each deck and it's position on the library. Originally it was planned to take advantage of the UNIX file system by making the directory a contiguous area starting at some very large block number on the library. Under the UNIX file system disk space is assigned, as required, only to those blocks of a file containing valid data, that is those that have been written on. This would have made the expansion of the directory a simple matter without posing any real limit on the size of the library, but unfortunately several problems were created which finally lead to the adoption of an alternate design.

The most obvious problem was that the size of a library, obtained through the ls command, would always be very large (and constant) even though the actual disk space required may be quite small. Secondly if a user tried to copy a library (using the cp command) then the resulting file would indeed use all the space reported by ls and not only that used by SLMP. Another more important

problem to be faced was how to read through the directory of a library which could virtually extend to any length. Decisions that had to be made were; what size buffer should be used; should a variable size buffer be dynamically assigned; and what conditions would cause the the reading and writing of a copy of the directory.

The alternative proposed was a circular chain of fixed length directory blocks created as required and interspersed with the source code decks stored on the library. As well, a library header at the beginning of the file was needed to accommodate a table of unused space on the library (produced by the deleting of decks) and a pointer into the chain of directory blocks. This allowed a fixed size buffer to be allocated and it was realised that the contents of the buffer only needed to be written out when another block had to be read in. The latter method was accepted because it imposed no restrictions on the size of a library and although some problems still existed they were more clearly defined and thus more easily solved.

The original idea for modifying a deck was to add to the front of each line of a deck a six character line sequence number. Modifications were to use a basic system of deleting existing lines and inserting new lines by referring to these line sequence numbers. New lines inserted in the deck were to be identified by the six character identifier supplied by the user. Associated with this idea was a plan to provide an option for resequencing the deck which implied replacing all identifiers by line sequence numbers again. Problems involved with the implementation of this concept led to the most important and basic feature of the package being completely redesigned. Some of these problems were:

(1) a satisfactory method of obtaining the original source code at any time had still not been found.

(2) how could an inserted line be deleted if it did not have an associated line number?

(3) how could a deleted line be excluded from the most recent version of a deck while still remaining in the original?

(4) the resequencing of the lines of a deck would mean forfeiting all information describing any previous updates to the deck.

All of these problems were overcome and a more easily maintained, more flexible, and more sophisticated package was produced as a result of one very simple idea. All other design features are an extension of this idea.

Firstly all lines of the original deck are stored exactly as they exist on the input file and always remain as part of the deck. When a new line is inserted in a deck two bytes are added to the begining of the line which have a threefold purpose. The first byte is the ASCII 'escape' character which identifies the line as affected by an update. The second byte contains a pointer to a table of identifiers which contains any information about the group of lines that are part of the complete update. Some information contained in this table is; whether or not the identifier

is still required on listings (the 'clear-flag'); whether the update is current or has been removed (the 'forget-flag'); and the time and date of the update (not implemented). Also contained in the second byte are two flags which indicate if the line has been deleted or inserted. Similarly when a line is deleted the two bytes are inserted (or modified). Therefore by inspecting the first two bytes of a line and the corresponding identifier table entry (if the line was part of an update) we can readily determine if the line exists in the most recent version of the deck. By the same method lines of the original deck can be found because they will either have not been modified or will not have the 'inserted-flag' set. Unfortunately a flaw exists in this design as once a line is deleted any connection with a previous update is lost. However, this can be overcome by implementing a variation of the design which is discussed in the section 'Future Developments'

The third major design decision made concerns the method of applying the updates to a deck. Three situations exist where modifications to a deck are allowable;

(1) 'listing' a deck.

This function creates a file containing a copy of the deck with line sequence numbers and update identifiers included. The previously discussed concept of applying modifications allows any one line of a listing to have a line number, a modification identifier and a flag (indicating whether the line has been inserted or deleted) all present at once. It was decided to give the user the option of including the identifiers and flags and deleted lines. The method of generating the line numbers introduces the concept of the 'curent deck'. Line numbers used in an update correspond to the lines of the current deck and therefore only those lines of a listing will have prepended line numbers. This means that deleted lines can be included in a listing without line numbers so that the user can be assured that the line numbers will always correspond to the same line (i.e. until another update).

(2) 'extracting' a deck.

This creates a file suitable for compilation in the same format as the input file required by the ^add and ^replace commands.

(3) 'updating' a deck.

The update function makes a permanent change to the deck on the library.

The desired goal was to be able to create the exact same result by

(i) listing (or extracting) a deck with a set of updates, or

(ii) permanently modifying the deck with the update function using the same set of updates and then listing the deck with no modifications.

To ensure this result all three functions firstly call a routine to create a temporary deck using any supplied modifications. This temporary deck is the same format as decks on the library and the update function simply replaces the existing deck with the modified one. The list and extract functions generate the required file by operating on the temporary deck as if it was permanent on the library. If no modifications are made the existing deck on the library is used avoiding the need to create an identical temporary file.

3. IMPLEMENTAION

3.1. File Format

At the beginning of every library created by SLMP is a library header. The purpose of the header is to identify the file as a valid library and link together the other components of the file. The structure of the header is

```
struct header
{
        int firstblk;
        int numfree;
        struct free freetab[FTABSIZE];
}
```

A magic number or string has yet to be implemented to protect against the misuse of SLMP on a file without the correct format. numfree is the number of blocks of free space on the library which are described by the first numfree entries of freetab. freetab is a table of 'free space' on the library and has the following structure.

```
struct free
{
        int start,fsize;
}
```

Free space is an area of the library that is currently available to be used. It is created directly by the ^delete command and indirectly by the ^replace and ^update commands. Each entry of the table contains the starting position and length of a contiguous piece of free space. The end-of-file of the library is maintained as a very large free space starting at the first byte after the last piece of valid data.

firstblk is required to locate the circular chain of directory blocks which contain one unique entry per deck on the library. Each block of the directory chain is a fixed length and is created as required when new decks are being added to the library. The structure of each block is

```
struct dirblock
{
        int used;
        int thisblk, nextblk;
        struct direntry dir[ENTPERBLK];
}
```

used is a flag indicating that the incore copy of a directory block has been modified and must be written to disk. Actual blocks on the file should never have this flag set. thisblk is a pointer to the position on the file where this block must be written and nextblk is the link in the chain and is the position on the library of the next directory block. dir is an array of directory entries, one per deck which have the following structure.

```
struct direntry
{
        int mods;
        char name[NAMESIZE+1];
        int  size;
        int  loc;
}
```

At present only the necessary fields have been implemented but information such as the time of creation and time of the last update would be useful in any future developments. All decks have a unique name and are stored as a contiguous block. The starting position and size of the deck are stored in loc and size respectively. The mod field contains the number of modification identifiers stored in a table which is located immediately after the deck and is included in the size. This table varies in length (calculated from the value of mod) and does not exist when a new deck is initially added. Each entry of the mod-table has the following structure.

```
struct modentry
{
        char id[7];
        char flags;
}
```

flags contains two flags which can only be set as a result of a corresponding update directive (-clear or -forget). The first is the 'clear-flag' and if it is set the corresponding identifier is omitted from any listings. The second is the 'forget-flag' and when it is set any modifications from the corresponding group are ignored, that is deleted lines pointing to the entry are part of the current deck and similarly inserted lines are not. Another field useful in future developments would be the time and date of the corresponding update. The id field is the six character identifier supplied by the user as the parameter to the -mark update directive.

3.2. I/O within SLMP

The function of SLMP is to maintain a library of source code decks and for this reason alone it is important that certain considerations be given to the I/O within the package. Because updates operate on a line basis, the lines of the current deck must be counted during most I/O operations. This of course requires the counting of newline characters which involves the checking of each character separately. Although this means some operations on a large deck could be very slow it is fairly simple and also appears unavoidable. Slightly modified versions of getc and putc have been implemented to suit the particular needs of SLMP. Separate buffers are used for the reading and writing of files and associated routines to create and open files also initialize these buffers. The structures of the buffers are

```
struct ibuf
{
        int cntdown, ifildes, nleft;
        char *nextp;
        char ibuff[BUFSIZE];
}


struct obuf
{
        int ofildes, nunused;
        char *xfree;
        char obuff[BUFSIZE];
}
```

Both buffers have corresponding fields for the file descriptor of the file being read or written and a character array for the buffer proper. nleft and nunused are the number of characters left in te input buffer and the number of unused spaces left in the output buffer respectively. Similarly nextp and xfree point to the next character to be read or written in the corresponding buffers. The input buffer has an extra field, cntdown. Normally when input is an entire file rchar returns a zero value if the buffer is empty and a call to read fails. If however input is a deck on the library cntdown is used to determine the end of the deck. rchar will return a zero value when the buffer is empty and cntdown is zero. When input is from a file cntdown is set to -1 to indicate that it is not being used, otherwise it is set to the size of the deck and decremented by the number of characters read whenever the buffer is refilled. If cntdown goes negative nleft is set to the appropriate value and cntdown is reset to zero.

Three different methods of I/O are used within the package.

(1) Copying an entire file directly.

When a new deck is added to the library via the ^add, ^replace, or ^update command the function newdeck is called to copy the input file directly to a suitable position on the library. newdeck opens the input file and stores the size in the current directory

entry. It then searches the free table for the first area of free space large enough to store the deck. When a suitable position is found it stores a pointer to the beginning of this area in the directory entry, updates the free table and calls copyfile which allocates a buffer and copies the new deck onto the library in blocks of BUFSIZE (512) bytes.

(2) one character at a time.

During a list, extract or update of a deck the first few characters of each line must be inspected separately to determine the status of the line with respect to the current deck. The two functons rchar and wchar are used to read and write respectively any number of characters to or from a given file (buffer).

(3) one line at a time.

When it has been decided if the current line is required in the new deck it can be copied onto the output file or skipped over. The function tilleol copies all characters up to and including the next newline character from one file (buffer) to another. Similarly the function skip2eol reads characters from the input file until the next newline. The importance for all decks to be source code becomes obvious here as both of these functions terminate with the newline character which is used to delimit lines of source code. No real check can be made when a deck is being added to the library but certain restrictions regarding the length of lines can be applied during a ^list, ^extract or ^update.

3.3. Brief Description

The mainline of SLMP has three major steps;

(i) prepare the library for processing;

(ii) process all the requests;

(iii) write out the header.

To process the requests the function getcom is called to obtain and validate the next command and any corresponding parameters. getcom in turn calls line to read the next line from the standard input. It then checks that it is a legal command under the mode of the call and sets up the required parameters. getcom only returns to the mainline when it finds an allowed command with the correct parameters. line sets up pointers to the begining of each group of non-blank characters and returns the length of the line to getcom. The appropriate function is then called to perform the required action. Any operation on a specific deck (i.e. all except ^table and ^pack) will cause a call to be made to the function getentry. This function searches the directory chain starting with the current block for an entry with a name matching the first parameter of the command. If found it sets the global variable dirptr to point to the corresponding incore entry and returns a value of 'true'. Otherwise it sets dirptr to the first unused entry found (having a null string in the name feild) and returns a value of 'false'. If an entry for the required deck is not found and no unused entries exist getentry creates a new

directory block and links it into the chain.

At present there are eight commands which can be split into three general categories. First there are the ^add, ^delete and ^replace commands that operate on a deck as a unit and are not concerned with individual lines. The second category includes the ^list, ^extract and ^update commands that process each line of a deck separately and accept an optional set of update directives from the standard input. Finally there are two miscellaneous commands, ^table and ^pack which have no parameters and operate on the library as a whole. The implementation of these commands is covered in the following section with some of the algorithms used being described in detail.

3.4. Detailed Description

3.4.1. ^add

This command is relatively simple to implement as it virtually is only a file copy. Firstly getentry is called to find the directory entry for the deck and a check is made for a duplicate deck-name. If the deck did not previously exist newdeck (described in the section 'I/O within SLMP') is called to copy the input file to a suitable location on the library. If this is sucessful the deck-name is copied into the directory entry, the mod field is set to zero and the used flag is set to make sure the directory block is written out to disk.

3.4.2. ^delete

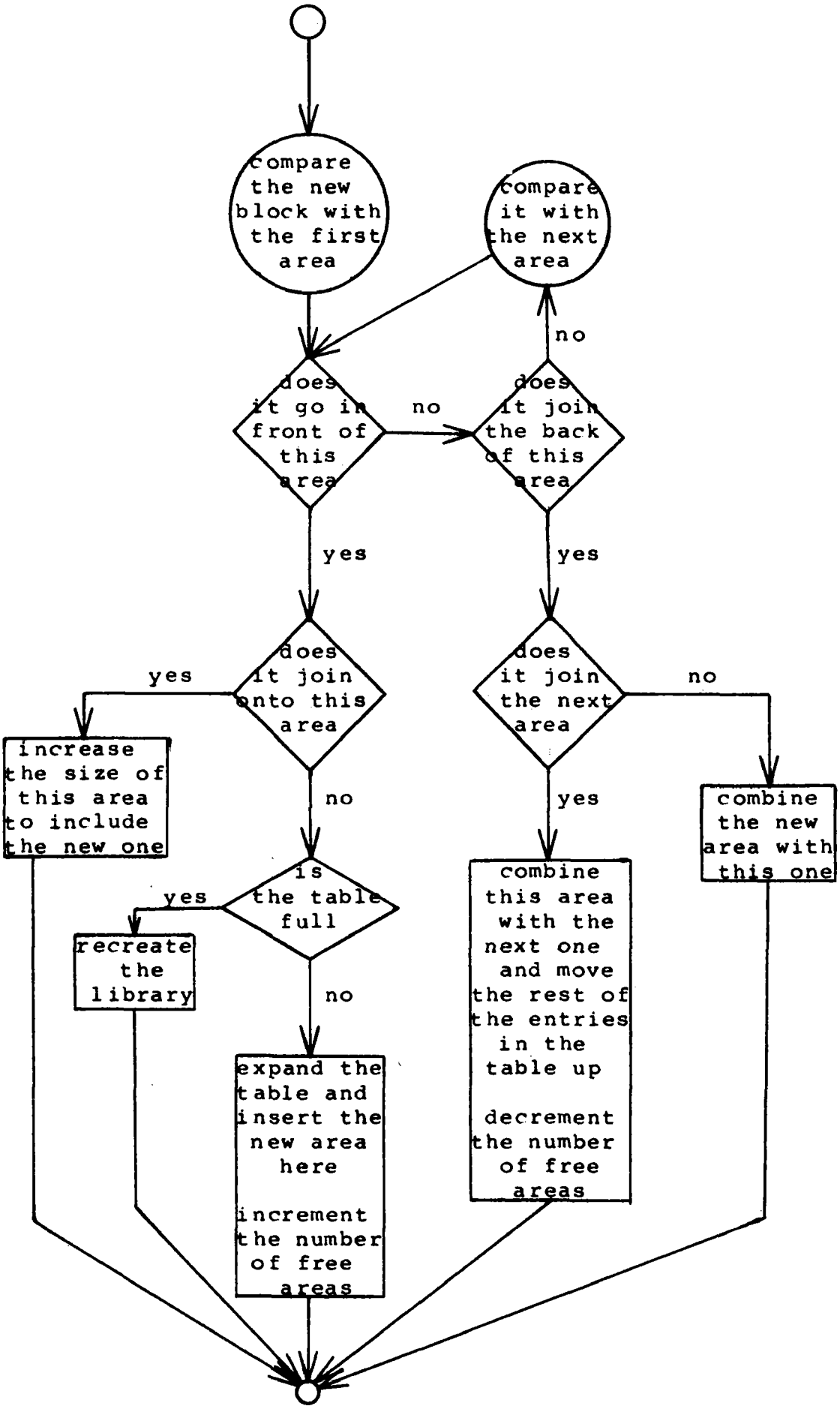
To delete a file the directory entry is marked as unused by storing a null string in the name field, and calling savespace to add the area where the deck was stored to the free table. Entries in the free table are stored in order of the position on the library so that two adjoining areas can be combined into one. If the new area being added to the table does not join any other blocks then a new entry must be inserted in the table. This could cause the table to overflow if the library has become fragmented in which case pack must be called to recreate the library and squash out any free areas. The user can also request that the library be recreated by using the ^pack command which is described later. The flowchart of the function savespace has been included here because although the idea is simple covering all possible cases complicates the task of determining the position of the new area in relation to already existing blocks.

3.4.3. ^replace

^replace is merely a combination of the ^add and ^delete commands with the condition that the existing deck is not deleted until the new deck has been successfully copied onto the library.

3.4.4. ^list

This is the command that provides the user with the line numbers to use in an update. It also identifies the lines affected by each update.



Flowchart for 'savespace'

December 19, 1978

There are three options which can be used with the ^list command - D, I and M. The M (mark) option causes the update identifiers (used in the -mark update directive) to be included in the listing if they have not been 'cleared' and if the update has not been 'forgotten'. The D (deleted) option will cause any lines that have been deleted to be included in the listing and flagged with a 'D'. These lines are not part of the current deck and do not have line numbers prepended to them. The I (inserted) option tells SLMP to flag any lines that have been inserted with an 'I'.

As mentioned before the ^list, ^replace and ^update commands all call the same function to apply any optional modifications. In fact there are two functions used to do this, getmods and moddeck.

3.4.4.1. getmods

This is the function that uses the update directives to produce a new mod-table and a set of modifications suitable as input to moddeck. It calls line to get the next line of input until either end-of-file is reached or another command is read. For each line it processes the update directive of which there are five. Three of these directives modify the table appended to each deck and the other two affect the lines of the decks. The directives are

(i) -mark which uses a six character identifier to mark the following changes to the deck. It stores the identifier in the table and remembers the pointer to that entry.

(ii) -clear sets up a flag in the entry of the table corresponding to the supplied identifier to unmark the update.

(iii) -forget sets a flag in a similar fashion to -clear to undo any corresponding changes. That is it effectively inserts any deleted lines and deletes any inserted lines.

(iv) -insert creates a new entry in a linked list of updates and switches getmods to input mode so that subsequent lines of text are stored on a temporary file. A line is considered to be text if the first character is neither a circumflex ('^') nor a dash ('-') and is stored in the same format as lines are stored on the library. The position of the first line and a count of the number of lines for each directive is kept in the list of updates.

(v) -delete works the same as -insert except that the entry in the linked list will also indicate that certain existing lines be excluded from the new deck.

The new mod-table and linked list of updates is input to the associated function moddeck which is always called immediately after getmods. The list of updates is sorted according to line numbers and the structure of each element is

```
struct changes
{
        int copyto, skipto;
        int insert, nlineloc;
        int mindex;
        struct changes  *next;
}
```

The use of each field is described in the following section.

3.4.4.2. moddeck

The output produced by this function is a flag (newlines) to say if any new lines have been inserted in the deck and an input buffer set up to read an already opened file which contains the new deck and the associated table. When the newlines flag is set what it really means is that the new deck is on a temporary file as opposed to the permanent library and it is set if and only if one or more -insert and/or -delete update directives were applied. Unless the list of updates created by getmods is empty moddeck uses this list and the current deck on the library to create the modified deck. Both the -insert and -delete directives produce the same type of entry in the list of updates.

The first field in each entry says copy any lines from the input deck until the line number of the current deck exceeds copyto. Similarly skipto is used to skip any lines (i.e. read only) that are to be deleted. The -insert directive sets skipto to zero so that no lines are skipped. The insert field is a count of new lines of text that are to be inserted after the necessary lines have been copied and/or deleted and nlineloc is the starting position of the corresponding lines of text on the temporary modfile created by getmods. mindex points to the entry within the table of identifiers which contains the identifier current when the directive was applied. Finally next points to the next update in the list.

The function cline is called by moddeck to copy or 'delete' (i.e. prepend or modify the two special bytes) the next line of the current deck. Any lines not part of the current deck are simply copied and then the next line is checked. There are three ways that a line can be part of the current deck:

(i) it is an original line that has never been affected by an update.

(ii) it has been inserted and the update has not been 'forgotten'.

(iii) the line has at some time been deleted and the update has been 'forgotten'.

3.4.4.3. ldeck

Ldeck is called by list to create the final deck with line numbers. It must inspect each line and determine if it is to be included in the listing and what flag and/or identifier if required has to be prepended.

3.4.5. ^extract

This function operates almost identically to list except that no options are processed and edeck is called instead of ldeck to create a file without line numbers and without deleted lines.

3.4.6. ^update

Update calls getmods and moddeck in the same manner as list and extract and then calls newdeck if necessary to replace the existing version. Moddeck will have created a deck on the temporary file in the format required to store it on the library and no further processing needs to be done by update.

4. FUTURE DEVELOPMENTS

As with any large program there always seems to be some way to improve the design and include new features which lead to a much more powerful system. Many changes were made to SLMP during its development and still more ideas have been suggested which were unable to be implemented due to a lack of time.

There is one idea that would provide a much more effective recording system for modifications and allow a copy of a deck to be obtained as it was at any time. To implement this would involve changing the method of identifying lines affected by modifications and increasing the amount of data stored in the table appended to each deck. Before discussing the idea we must realise that to be stored on a library a line must be part of an original deck or must have been inserted during an update. Also although a line may only be inserted once it can be deleted any number of times because when a modification is 'forgotten' all deleted lines appear again. The idea then is to provide a method of having a variable length header on each line with pointers to every modification identifier in the table that is connected with the line. The mod-table would also be expanded to include the date of the modififcation and also if it has been 'forgotten', then when. This system would supply all the information required to determine the status of a deck on any given date.

5. BUGS

Although the system is operating successfully there are many different situations that may occur where the data on a library may become corrupted. The program needs to have several precautions included to try and minimise the number of situations which could create a suspect library, and a recovery routine should also be written.

A corrupt library is most likely to be caused by problems internal to SLMP or by an I/O error but user's errors may result in a deck not containing all the information expected. Although most

of these situations are adequately checked error messages may appear some time after the error has been made giving a false impression of what was really wrong. This is definitely the case when input is redirected from a file because the error messages do not specify which command or update has failed. This is another area that does require some improvement.

A third situation which goes unchecked at the moment is the use of invalid files (i.e. files of the wrong format) either as a library or a source deck. Both these situations need to be controlled and will almost certainly cause SLMP to blow up if they are not.

6. CONCLUSION

The package that has been produced upto now can be used very effectively until the level of modifications becomes too deep. It provides a useful system for recording modifications made to a deck and clear listings readily showing the current state of a deck. Although the package is at present very susceptible to misuse with a little work it could be made more secure and with the addition of the new method of recording modifications it could be used as a very powerful tool.