SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 479–489 (1981)

The Construction of a Portable Editor

J. E. L. PECK
Computer Science Department, University of British Columbia, Vancouver Y6T 1W5, Canada

AND

M. A. MACLEAN
Computer Science Department, University of Canterbury Christchurch 1, New Zealand

SUMMARY

The paper describes the implementation of an editor designed and constructed with portability in mind. We give first a brief introduction to the editor from the user's point of view, then we discuss the methods used to ease the problems of portability, and finally we explain some data structures used in its implementation, structures which, we believe, are interesting in their own right.

KEY WORDS Portability Editor System software

INTRODUCTION

This paper discusses the implementation of a software engineer's editor called CHEF. To understand the details it will be necessary to say something of how CHEF appears to the user, but a detailed user's manual is not intended. That, we hope, will appear elsewhere.[1] Then we shall discuss the steps we took to ease portability, for we consider this to be a contribution to the art of producing system software. The main thrust of the paper, however, is a description of the internal data structures used by CHEF showing how their form is influenced by the operational requirements. We are indebted to W. E. Webb for the BCPL implementation of a precursor to CHEF, but he may not recognize the final product.

THE USER'S VIEW

The editor is called CHEF, a name which stands for the Christchurch Edit Facility. It is a happy coincidence that CHEF also means a superior preparer of good things. From the user's point of view CHEF will appear somewhat like the editor described in Software Tools by Kernighan and Plauger,[2] or like the UNIX editor[3] known to so many users of that happy operating system. Since we shall be referring to this latter editor frequently we shall call it ED. That there is a similarity between ED and CHEF is not surprising, for we acknowledge with gratitude our indebtedness to the authors of ED for the elaboration of many principles of good design. We hope, however, that we have progressed further, in that we have applied the principle of orthogonality, i.e., we have reduced the number of independent concepts to a minimum and we have tried to increase the power of each to its maximum. We have also added some new features not

0038–0644/81/050479–11$01.00
© 1981 by John Wiley & Sons, Ltd.

Received 4 March 1980

found in ED, but possibly existing in other editors or text processors. We hope that the total assemblage will now offer a power and utility that will make CHEF attractive and appealing to users.

NEW FEATURES

One new feature added is for simple word processing by built-in rules, including centering and right justification of text. Some may consider this to be a frill for an editor intended for the programmer's workbench, but it may turn out to be a boon to the busy software specialist with minimal secretarial help and to those who like to see well written comments neatly displayed in source programs. It could also make CHEF useful to the general user as a simple and flexible word processor. Another new feature added is a number of one-line text buffers, which we call 'controls'. These are for storing text, patterns, commands, parts of commands or sequences of commands for later use. Another new feature, unavailable in ED, is an ability to stack work-spaces. This means that CHEF may suspend the editing of one file, to edit another, without losing the work-space and status of the former. One might argue that ED can do this with '! ED F', but this is a happy consequence of the recursiveness of the UNIX shell and not a property of ED itself. Like ED, CHEF works on a copy of the original file and never changes the original until the user gives the word.

As we have already mentioned, CHEF is similar to ED. In particular, its commands all have the same syntax, *viz*. a location followed by an operator, followed by a suitable operand. The location is usually one line or a range of lines. We have arranged that there is only one default location, which is the current line, and in this respect we have a simplification of ED. We have added the possibility that operators (which are one letter) may be qualified by a one-letter suffix (or modifier). This permits a unification of concepts. For example, the operator *P* stands for print (as in ED), but *PN* prints the line number, *PA* (print all) prints the line number and the text, *PC* prints the byte count, *PL* prints lucidly, i.e. unprintable characters appear in octal, and *PFx* prints to the file *x*. In ED each of these is handled by a different operator.

ED	CHEF	Meaning
1, 2*T*7	7*I*1, 2	Beyond 7 insert lines (1,2)
7*R x*	7*IFx*	Beyond 7 insert file x
1, 2*M*7	7*ID*1, 2	Beyond 7 insert lines (1,2) and delete the source
7*A*	7*I*	Beyond 7 insert from the console
7*I*	7-*I*	Before 7 insert from the console
1, 2*C*	1, 2*C*	Change (1,2) to text from the console
1, 2*D* 5,6*M*0	1, 2*CD*7, 8	Change (1,2) to text from lines (7,8) and delete the source
1, 2*D* 0*R x*	1, 2*CFx*	Change (1,2) to what comes from file x
1, 2D 0'*T*7,8	1, 2*C*7, 8	Change (1,2) to text from (7,8)

Another example of unification of concepts concerns the movement of lines of text from one position to another. In CHEF there are the operators *I* (insert) and *C* (change). Both may take a suffix modifier *D* (delete) and both may have a location or a file name on the right. This allows us to express more operations than can be expressed with six operators in ED (*T*, *R*, *M*, *A*, *C*, *D*,). A comparison of the two at this point may be useful.

As can be seen in these examples, all CHEF commands that modify the work-space place the destination location consistently on the left of the operator and the source location on the right. In ED there is a confusion of directions, in that both *M* and *T* take a destination on the right whereas, for the others, it is on the left.

PORTABILITY

CHEF is written in BCPL. The fact that the BCPL compiler is itself written in BCPL[4] and is available on many machines gives us a large measure of portability from the start but, of course, there is much more to portability than that.

The version of BCPL that we use (BCPL–V, V for Vancouver) is derived from the original Cambridge University version. It has enhancements found in many other BCPL compilers and some home-grown improvements that we find very helpful and that now form part of the proposed BCPL standard.[5] There are two improvements, in particular (for which we are indebted to V. S. Manis), which we have found especially useful in constructing such system software. They are

(a) a byte subscripting facility,
(b) Conditional compilation.

The byte subscription facility can best be described by saying that the BCPL statement usually written

```
putbyte (s, i, getbyte(t, j))
```

is with our compiler expressed as

```
s%i := t%j
```

Not only is this more expressive of the programmer's intention, but it can lead to more efficient code on some machines.

The second improvement which we use is conditional compilation. By this we mean that our compiler checks every expression appearing as a condition to see whether it might be a constant expression. If this is the case, then code for that part of the conditional command or expression, which will never be visited, is not emitted. For example, the command

```
TEST 1 = 2 THEN c1 ELSE c2
```

will produce exactly the same code as will the command *c*2 alone.

The byte subscripting facility has meant much to us in the development of clear and understandable source code in the many character manipulating procedures such as for pattern matching. Conditional compilation has been a crucial factor in allowing for portability and adaptability.

ONE SINGLE SOURCE FILE

It is not new to observe, but is worth stressing again, that a portable system intended to run on several different machines must have only one file of source code. Any other method becomes far too unwieldy not only during the process of system creation, but especially during the maintenance phase. We thus use conditional compilation to ensure that there is only one file of source code for all versions of the editor on all machines. Moreover, the code automatically checks that every section has been compiled compatibly.

To demonstrate that conditional compilation can increase the adaptability and ease the burden of maintenance, we give below a procedure from the machine dependent section of the source code, which is concerned with setting the file associated with the stream *s* to the position *n*. This position could be a line number or a byte number depending upon the system.

```
LET set_file_position(s, n) BE
{1 IF sys_aos THEN
   { LET pos = VEC 1 AND err = 0
     pos ! 0, pos ! 1 : = 0, n
     err : = setposition(s, pos)
     IF err  <0 THEN stop(err)}
   IF sys_mts THEN s ! linenumber : = n * 1000
   IF sys_fmgr THEN
   {HP LET ir = 1
     call(@sys_posnt, 4, s ! dcb, @hperr, @n, @ir)
     IF hperr  <0 THEN err("Set s = °0n, n = °0n", s, n}HP
   IF sys_rdos THEN set_pos(s, n)
   IF sys_unix THEN seek(s, n, 0)}1
```

In the above routine each of the conditions '*sys_aos*', '*sys_mts*', '*sys_fmgr*', '*sys_rdos*', and '*sys_unix*' is a manifest constant, only one of which is true. In order to change the source code to suit a particular machine it is only necessary to change the truth values of these constants. This is done by altering the definition of one manifest constant in the header.

One might argue that, if BCPL implementations on various machines were what they should be, then there would be a complete set of standard interface routines able to handle all users' requirements such as the above, and the special treatment of each machine in a particular piece of system software would not be necessary. Unfortunatley this ideal situation does not now exist. We all look forward to the day when it will and the proposed BCPL standard[5] is a step in that direction. Meanwhile we are able to overcome these deficiencies neatly by keeping similar code for different machines close together in the source code knowing that there is no extra penalty in proliferated object code.

MENU SELECTION

Another of our concerns was to allow the editor to be tailored to suit both large and small computers, with suitable reduction in the use of memory. This is accomplished in several ways, but one of the important ways is to allow the editor's implementor to select, from a menu of features, those to be included. This selection is done by changing the definition of an appropriate manifest constant from TRUE to FALSE, to disable the code for a particular feature. As an example, CHEF has an undo command, *U*.

Accordingly there is, in the header, a manifest constant '*menu_u*', which for the full editor is set to TRUE. If one believes that the undo feature is not worthwhile, or if there is not enough memory to accommodate the code for it, one may set this constant to FALSE, whereupon no code will be generated for the undo feature.

THE DATA STRUCTURES

To understand the data structures we first note that there are essentially four places (Figure 1) where data is located during an editing session. Firstly, there is the user's console. Here editor commands are entered or lines of new text are created. Secondly, there are the files belonging to the host operating system which the editor must be able to open, read, write, position and close. Thirdly, there is a 'text-place' in memory through which whatever line of text we are dealing with passes, whether it be a command or a piece of text to be edited. (We shall often hyphenate words which are given a technical meaning.) Fourthly, there is a 'work-space' in which lines of text are held waiting for manipulation by the editor. It should be clear from figure 1 that all lines of text must pass through the text-place.

```
+-----------+                             +------------------+
|           +--------.         .----------+                  |
|           |   get  |         |   read   |                  |
|  console  |        |         |          |      files       |
|           <-----.  |         |  .------->                  |
|           | put |  |         |  | write |                  |
+-----------+     |  |         |  |       +------------------+
                  |  |         |  |
            +-----+--V---------V--+-----+
            |        text-place         |
            +-------A-----------+-------+
                    |           |
               fetch|           |store
                    |           |
          +---------+-----------V---------+
          |                               |
          |           work-space          |
          |                               |
          +-------------------------------+
```

Figure 1. Simple view of data flow

The terminology used in the code helps to identify the parts of the system involved in transfer of data. For example, all routines which are concerned with input from the console or output to the console start with '*get*' or '*put*', such as '*put_msg*', for displaying error messages at the console. Similarly the words '*fetch*' and '*store*' are used in the transfer of data between the work-space and the text-place. Transfer of data to and from the system files is signalled by use of the words '*write*' and '*read*' in the identifiers of the routines concerned with it. We believe that a consistent use of words in this manner will contribute much to the understanding of the source code.

THE EDITING OF LARGE FILES

Many editors on small computers soon run into the problem that there is insufficient memory to hold both the editor code and the text of the files to be edited. To overcome this, some editors resort to a method of 'windowing', by which we mean that only a portion of the file to be edited is available at one time. We have used editors in which the window could only be moved forwards and it was not possible to go back to line 1 of a large file without completing the editing process and starting again. Other editors have two sets of editing commands, one for small files where no windowing occurs and another to deal with large files where a window technique is the only way to accomplish the task.

We feel that both of these methods are unacceptable. The user should not have to put up with a severe performance limitation and neither should he have to learn two sets of commands. Although we accept that windowing is inevitable, we have aimed to make the user unaware of it. We feel that the software should take care of the details while the user is free to jump from the first line of his file to the last, and back again, even when he is using a minicomputer with small fixed memory size.

There are two potential sources of difficulty in editing large files. The first is the number of characters in the file and the second is the number of lines. We will show how data structures can be devised to avoid both these pitfalls and allow CHEF to edit very large files with surprisingly small commitment of memory. In fact, the only arbitrary limitation is that the line number of the last line of the file should be an integer capable of fitting into one memory cell.

THE WORK-SPACE

We have already identified four places where data may reside: the console, the text-place, the files and the work-space (Figure 1). From the point of view of the user, and indeed for the major part of the editor's code, the work-space is merely a sequence of lines. There is a 'work-space-manager' which carries out the deletion, insertion and copying of lines. This manager is an autonomous section of code which, in accordance with good programming practice, alone knows the details of the implementation of the work-space. Communication with it is essentially through the routines '*fetch_line*', '*store_line*' and '*copy_lines*' whose parameters are one or more line numbers.

The first factor to be considered is the number of characters in the file being edited. Clearly only small files can be completely stored in memory: a 500-line file with 70-character lines would consume over half of the memory capacity of a minicomputer with 64K bytes of memory. So we must consider the use of disc storage and provide a means to locate random lines of text within it. This we can do either by storing each line in a fixed sized segment of the disc storage so that its position can be calculated from the line number, or by packing the text closely in storage and providing a vector of line pointers for access to individual lines. We use the second method to avoid excessive waste of disc space and the need to move text within the work-space during editing.

Although disc capacity does not set an unacceptable limit to the size of the work-space, the word length of the machine limits the size of each line pointer. Using once again the example of a 500-line file with 70-character lines, the pointers need to be as large as 35,000 which is beyond the capability of a 16-bit machine using two's complement representation.

We have solved this problem very simply by aligning the start of each text line in the work-space with discrete addresses spaced a fixed number of bytes apart. Thus we reduce the number of bits needed to specify a line pointer while only wasting a small amount of space between records. The spacing can be chosen by the implementor to suit his machine. For a 16-bit machine, 4 bytes is an appropriate spacing and shortens the required size of a line pointer by two bits, enabling a file of 128K bytes to be addressed.

In our terminology we refer to the storage place for the text records as the '*record-place*', to the byte multiple as a '*grab*' (the unit of data in the record-place for addressing purposes), and to the vector of line pointers as the '*pointer-place*' (see Figure 2)

```
                  +----------------------+
                  |abc                   |
                  +----------------------+
                        text-place

+-------------------------+---+-------------------------------
|                         | o |
+-------------------------+-+-+-------------------------------
<----------- i ---------->  |      pointer-place
      v---------------------'
<-----+---->
+----------+---+----------+----------+------+--------------+--
|          |abc|          |          |      |              |
+----------+---+----------+----------+------+--------------+--
                                  record-place
```

Figure 2. Layout of work-space (showing access to i*-th line)*

MOVEMENT OF LINES WITHIN THE WORK-SPACE

When a file is edited, a work-space copy is made. If the file has 10 lines and a line is now inserted, say after line 7, then its record becomes the 11th record of the record-place (not the 8th), but the 8th entry in the pointer-place will now remember the grab position of this record, after shifting the original 10th, 9th and 8th cells up by one. Deletion of lines is simpler; the entries of the pointer-place are merely shifted down by an amount equal to the number of lines deleted, and the unwanted records remain the record-place. This disregard of garbage collection in the record-place may seem wasteful, but we will soon see that it is not and that it turns out to be useful in implementation of the undo feature.

If lines are copied or moved, then their records in the record-place are untouched, the only changes occurring in the pointer-place. This means, for example, that if a line is copied to another position in the file, then there will be two entries in the pointer-place pointing to the same record. If the text of a line is altered, then a new record is made which is stored at the end of the record-place and the old record is no longer pointed to, except in the case of copied lines just mentioned where there might still be an entry in the pointer-place which points to the old record.

It can be seen that the two editing operations that require the most work to accomplish are the insertion and deletion of lines. In the extreme, as for example when a new line is inserted at the beginning of the work-space, the pointers of all the existing lines have to be repositioned in the pointer-place. The need to achieve efficiency in these two operations has had a strong influence on the method of implementing the pointer-place.

IMPLEMENTATION OF THE RECORD-PLACE

The record-place is implemented as a random-access disc file with block transfers of data to and from memory as needed. We call this file the 'record-store'. When the *i*th line of the work-space is needed, its pointer is retrieved from the pointer-place and divided by 'block gsz' which represents the size of a disk block measured in grabs (this is a constant set by the implementor). The quotient is the block number within the file and the remainder is the offset of the start of the desired record within the block. Of course the record may straddle two or more blocks of the file.

We keep a single block in memory together with a flag that indicates whether or not the block has been altered since it was first created or read from disc. We have used block sizes from 128 to 1024 bytes, with satisfactory results in all cases.

IMPLEMENTATION OF THE POINTER-PLACE

An important goal is to avoid an arbitrary limit to the number of lines in the file being edited. If such a limit must exist, let it be a very large one. Clearly, if the pointer-place is a memory-resident vector, any limit that leads to economy of memory use is likely to be unacceptably low from the user's point of view. So we are led to consider the use of disc storage for the pointer-place also.

Our first approach worked well for a year until we discovered its weaknesses and abandoned it for an improved method. It may be of interest to record both our failure and our success.

In approaching our first solution we noted that most editing action is sequential or takes place within a small region of the work-space. We therefore implemented the pointer-place as a memory-resident window of fixed width that could move up or down the pointer-place vector as required. Parts of the vector not in memory were stored in a disc file. The width was made reasonably large (for example about 500 lines) so that there was a high probability that line pointers would be in memory when needed. This window was subdivided into 'frames' of about 60 pointers which were the storage units for swapping purposes.

For most operations of the editor this mechanism is satisfactory. However, if a large number of pointers have to be moved a distance greater than the width of the window, it becomes intolerably slow. A good example of this is when the editor is required to delete the first 600 lines of a 1000-line file. To do this, the pointers to lines 601-1000 must be moved down to positions 1–400 in the pointer-place. If the window is wider than 600, well and good. Otherwise the window must move upwards to fetch each pointer and down again to store it, and this must be repeated 400 times.

The improved method is based on the recognition that the desirable working set consists of two groups of contiguous line pointers, not one, and that these groups may have to be separated by an arbitrarily large number of lines. There is no reason for the groups themselves to be particularly large and one implementation works well with groups of only 64 pointers.

The actual implementation is a slight elaboration of the mechanism used for the record-place. The pointer-place resides in a disc file and two blocks are resident in memory at any time, with a simple form of 'least recently used' algorithm to determine which block should be swapped to disc when a new one is needed. We refer to this file as the 'pointer-store', It was gratifying to find that the code required for this more efficient solution was less than that needed for the earlier one.

Using this data structure, good performance is obtained with a total buffer area of three 64-cell blocks in a 16-bit minicomputer, one for the text records and two for the line pointers.

THE UNDO FEATURE

For an editor that makes a work-space copy of a file, an undo feature is perhaps a luxury. Indeed, for many implementors, it could be the first menu item to be sacrificed for the sake of smaller code. The implementation, however, comes surprisingly easily, if we are prepared to undo only one command at a time. Every command that moves or copies lines, but does not alter records, has an easily computed inverse; e.g. the inverse of 4*ID*7,8 is 8*ID*5,6. Thus, for such commands it is only necessary to compute and to save the inverse command in case it is needed, and to execute that as the undo command. For commands that actually abandon records, e.g. *D* (delete) or *R* (replace), we have already observed that the abandoned record remains in the record-place, but that its pointer is deleted from the pointer-place. All we need do then is to keep a backup copy of the relevant portion of the pointer-place. This then needs another random access file called the backup-store.

To implement the undo feature in this way, we keep a short string in memory, called the '*trail_line*' where the inverse command is built. We also postulate an additional command *B*(back substitute) which is unavailable to the user. If *B* has no modifier, then the relevant pointers in the pointer-place are restored from their backup counterparts. If the *B* operator has an *I* modifier, then space must be made in the pointer-place for pointers from the backup store. For example, the inverse of 7,10*D* is 7,10*BI*. This means that the pointers for lines 7 and above will be shifted up by four to make room for the restoration of the pointers to the abandoned lines from the backup store.

Commands are classified, with respect to the undo operator, as willing, neutral and unwilling. As examples, *D* (delete) and *R* (replace) are willing, *W* (*write*) and *P* (*print*) are neutral, while *E* (*edit*) and *N* (*new*) are unwilling. A neutral command is one that does not disturb the work-space. The last willing command can then be undone, provided that only neutral commands have come in between.

MERGING OF STORE FILES

We now have three random-access disc files, the record store for storing text lines, the pointer-store for pointers to these lines, and the backup store for pointers saved for use by the undo operator. Such proliferation is undesirable because of the extra code involved in administering so many files. So we have merged them into one file.

The problem here is that each of the three aggregations of data occupies a variable space and yet we must be able to retrieve a particular data item from a calculated position within the combined file. Now the pointer-store and the backup-store can easily be made of the same size and structure, in fact it simplifies the code to do so. Then we note that there is a rough proportionality between the file space used for the record-store and that used for the pointer-store, depending on the average line length. Tabulating the ratio of blocks used for pointers to those used for text for various line lengths and word sizes, we have:

average line length	16-bit word	32-bit word
20 chars	0·1	0·2
40	0·05	0·1
60	0·03	0.06

Accordingly we have one store file in which blocks of the three types are interleaved in a fixed ratio. Currently we use one pointer block and one backup pointer block to every 6 text blocks, making a module of 8. There is nothing sacred about this ratio. In fact the code uses a manifest constant which may be changed by the implementor. The actual numbering of blocks is as follows:

file block number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
pointer block no.	0								1								2	
backup block no.		1								2								3
text block number			0	1	2	3	4	5			6	7	8	9	10	11		

The translation of the text block number '*t*' to the merged file block number *m* is therefore

$$m = (t/6)*8+t\ REM\ 6+2$$

which means that a work-space with very long lines will waste a maximum of 2/8 of the space available while a work-space of very short lines will have a longer store file than otherwise necessary.

INSERTING LINES INTO THE WORK-SPACE

It was mentioned earlier that the insertion and deletion of lines are the two editing operations likely to require the most computing time. In the case of deletion, the need to move pointers over a large distance within the pointer-place vector requires that two separated regions of the pointer-place should be in memory at the same time.

In the case of insertion, we must consider the exact manner in which insertion is carried out before we can optimize the data structure or the algorithm. When line are being inserted into the work-space from the editing console, it is impossible to predict exactly how many lines the user will type. When the inserted lines come from a second file it is inconvenient, though not impossible in principle, to count the lines before insertion. Our first insertion algorithm was primitive and simply made room in the pointer-place for one line pointer as each line was read in. Consequently, for each line, the pointers of all higher numbered lines in the pointer-place had to be moved. This was acceptable when lines were being entered from the console but impossibly slow if a large file was inserted at the beginning of an already large work-space. Unlike the deletion operation, the upward shifting of pointers is not much helped by the presence of two blocks of pointers in memory.

Our solution to this problem is to abandon the one-at-a-time shifting of pointers and to shift them in blocks instead. If we assume that a block holds 64 pointers, at the beginning of the insertion process we shift the pointers upwards in the vector by 64 places to make room for a possible 64 lines until another move of 64 places is necessary. When insertion ends, any unused pointer space is closed up. Although the insertion of a single line is now a more complex operation than before, this does not appreciably affect response time at the console. However the process is more efficient with larger insertions and dramatically so when large files are inserted. The upward shifting of pointers by 64 places is a simple matter of reading blocks from the disc, renumbering them, and writing them out again.

WORK-SPACE STACKING

We have not yet discussed the stacking of work-spaces. When a user temporarily suspends the editing of a work-space by calling for a new one the command is *N* (*new*). The manner in which the pointer-place is set up for this is indicated in Figure 3. The

```
-27         0 1          (line_base)100 101                 150
+-----------+-+-----------------------+-+-------------------+-
|           | |                         |                   |
| controls  | | work-space 1 (100 lines)| work-space 2 (49) |
|           | |                         |                   |
+-----------+-+-----------------------+-+-------------------+-
```

Figure 3. Layout of the pointer-place

number of the last line of the original work-space is known and is in the cell '*last_line*'. Another cell called '*line_base*' (initially zero) is kept. When the current work space is stacked, then the value of '*last_line*' is added to '*line_base*'. In the new work-space the access of line number '*i*' is done by first incrementing '*i*' by the value of '*line_base*'.

Status information for a work-space being stacked, its file name, current line and so on, is stored, as a record, in the record-place before the new work-space is set up. On unstacking a work-space the disc storage used by its text records and line pointers is recovered and then the status of the previous work-space is restored.

THE CONTROL LINES

It was mentioned above that CHEF allows for certain text buffers or 'controls., where lines of text or commands or patterns may be held for later use. There are 27 controls which are accessed by a letter (either upper or lower case) or the character '+'. For example the first control may be filled with text from the console by using the command @AC (change) and the content of that control may be recalled (as a macro expansion) by using %*A* anywhere in the command line. To accommodate the controls, there are 27 cells in the pointer-place which are accessible with a negative index (Figure 3).

Storage of controls is handled by the same mechanism as for text lines except that, when a control is altered, the new content is stored in the same place as the old. For this reason the alteration of a control cannot be undone. Records for the 27 controls are stored at the beginning of the record-place and we must allow for each one to have its maximum size, which is the same as that of the text-place.

CONCLUSION

This has been a description of a versatile programmer's text editor implemented in BCPL with special care taken to ensure that the code is adaptable to many systems and can be ported easily. The editor is already running under four operating systems, on an Amdahl 470 under MTS, on a Data General Eclipse under RDOS, on a Data General Eclipse under AOS and on a PDP11 under UNIX. The implementation is such that the editor's generator may select those features wanted, or perhaps, more importantly, may cut the editor down to size to fit his system, by selecting from a menu of features for inclusion. Implementation on a variety of computers with both small and large memory poses special and interesting problems for the implementation of the work-space. We have concentrated on those special problems and their solution.

REFERENCES

1. M. A. Maclean and J. E. L. Peck, 'The CHEF edit facility', *TM 80-1 Computer Science*, UBC, Vancouver.
2. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison Wesley, 1976.
3. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, Jr., 'Document preparation', *Bell System Journal*, **6**, 2115–2135 (1978).
4. M. Richards and C. Whitby-Strevens, *BCPL, The Language and its Compiler*, Cambridge University Press, 1979.
5. M. D. Middleton, 'A proposed definition of the language BCPL', University of Regensburg, 1979.