RED
A Better C Screen Editor, Part I

RED is a descendant of the public-domain text editor (called ED2) which I described in the January 1982 issue of *Dr. Dobb's Journal*. I started work on RED so that I could edit files without worrying how large they were. RED's most important new feature is a set of new buffer routines which remove all restrictions on the size of files which may be edited.

RED appears just the same to the user as does ED2. The differences between RED and ED2 are all under the covers, and it is on these programming details that this article will dwell. Because of the length of RED's listing, this article will be split into two parts. This month I'll discuss how RED's buffer routines work. These routines use a programming technique called *virtual memory*, which I'll explain in detail. I'll also discuss a new method for recovering from errors which saves a lot of coding. Part I of this article concludes with a discussion of the performance of RED's new buffer routines.

The new buffer routines require unbuffered, i.e., sector-at-a-time, I/O routines. Rather than write my own, I decided to modify the assembly language portion of the BDS C run-time library. Leor Zolman, the author of BDS C, has generously given us all permission to use these routines in any way we choose. This new library will be presented in greater detail next month in Part II of this article.

New Buffer Routines

RED'S new buffer routines really got started when I bought (for 8 dollars!) a copy of the "Just Like Mom's Editor" from the BDS C Users' Group. The editor is still available and is in the public domain. I would like to thank Scott Fluhrer and Neal Somos for their work on this editor.

RED's buffer routines are direct descendants of the buffer routines that Fluhrer and Somos wrote. After a few hours work, I converted their routines to work with the BDS C version of RED. I quickly discovered two problems with the code. First, though files larger than memory could be handled, there was still a limit on the size of the files. Second, the buffer routines did not keep enough information about the location of the current line. As a result, scrolling the screen was terribly slow.

This was the start of what has become four new versions of the buffer routines. I used the excellent BDS C compiler for the first three versions. For the fourth version, I converted the buffer routines from BDS C to Small C. Most of the conversion work consisted of changing structures to parallel arrays.

Now let's see how the new buffer routines work. The original file being edited, the *user file*, must be copied to the buffer before editing can be done. When editing is complete, the save command or resave command is used to copy the buffer back to the user file.

ED2's buffer routines kept the buffer entirely in memory. RED's buffer routines keep most of the buffer on a disk file, the *work file*, which is used only by the editor; the work file is created when the buffer is initialized and it is erased when the editor ceases execution.

The work file is made up of a sequence of fixed-length *blocks*. it is not just a sequence of characters. A block is a multiple of the sector size of the disk. (The READ_SIZE constant gives the number of sectors per block.) The new buffer routines use the following three operations on the blocks and sectors of the work file:

```
sysseek (fd, block)
int fd, block;
```

Positions the file whose descriptor is fd to the indicated block. The next read or write operation will be to the block. Sysseek() can also be used to extend the file, i.e., to increase the size of the file. If the block number is exactly one larger than the number of the last block of the file, the file is made larger. Some operating systems do not allow files to be made larger dynamically. They require that the size of the file be known in advance. However, many operating systems (including Unix and CP/M) do allow files to be extended in this way

```
sysread (fd, buffer)
int fd;
char * buffer;
```

Reads the current block into the block-sized buffer.

```
syswrite (fd, buffer, n):
int fd,
char * buffer;
int n;
```

Writes n sectors from the buffer to the file. When writing to the work file, n is always READ_SIZE. Thus, only full blocks are ever transferred to and from the work file.

The operating system does not have to provide exactly these primitives, but the operating system must make it possible to write them. This is the reason for the new run-time library in the Small C version of RED. Next month I will discuss these routines in more detail.

Inserting and Deleting Blocks

The CP/M operating system, like most operating systems, does not allow blocks or sectors to be inserted into the middle of a file or deleted from the middle of the file. CP/M does allow files to be extended after the file has been opened, but only at the end of the file. Thus, when sysseek() extends the file, the newly created sectors must follow immediately after the last sector which has been allocated.

Fortunately, these restrictions do not really prevent the editor from inserting and deleting blocks where required. For most purposes, the buffer routines can ignore the actual location of each block. The buffer routines link together the blocks in a list. As far as the buffer routines are concerned, the first block in the list *is* the first block of the work file, regardless of its actual location.

Each block contains two pointer fields: d_next, which points to the next block on the list, and d_back, which points to the previous block. Each pointer is an actual disk address; it is used directly by the sysseek primitive. A negative pointer indicates the end of the list. Two global variables, b_head and b_tail, point to the first and last blocks of the list.

Figure 1 (page 35) shows a typical list of blocks. Links are shown as arrows, except that negative links are shown as a slash. The values in the link fields are not

by Edward K. Ream

Edward K. Ream, 1850 Summit Avenue, Madison, Wisconsin 53705.

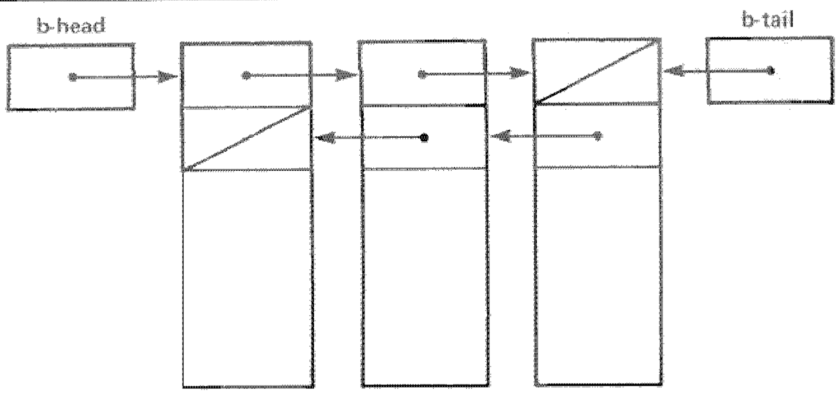
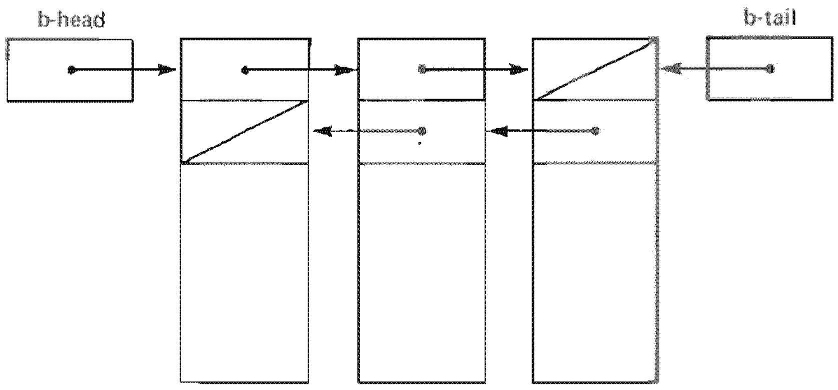
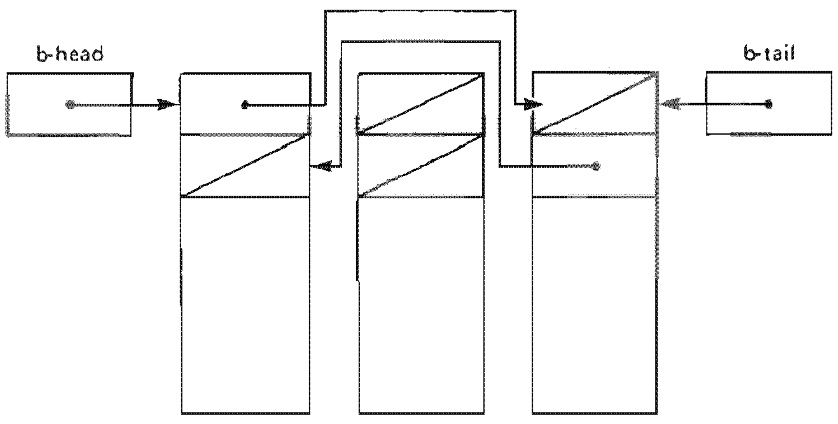


Figure 1.
The blocks of the work file form a double-linked list.



Deleting blocks: before
Inserting blocks: after

Figure 2a.
Inserting a block in the work file is just the reverse of deleting a block.



Deleting blocks: after
Inserting blocks: before

Figure 2b.

shown because the actual location of each block on the disk does not matter. The blocks of the work file are kept on a doubly linked list. This makes inserting and deleting blocks much easier. Searching the list backwards is also made possible.

Figures 2a and 2b (at left) show how to insert or delete blocks from the work file. To delete a block, the pointers to the preceding and following blocks are changed so that nothing points to the deleted block. Figure 2a shows the work file before the block is deleted and Figure 2b shows the work file afterwards. Inserting a block is essentially the reverse of deleting a block. Considering Figures 2a and 2b in reverse order shows how a block is inserted into the work file.

Deleting the last block of the work file is a special case, illustrated in Figures 3a and 3b (page 36). However, it turns out that the first block of the list never changes. It is never deleted, nor is a block ever inserted before it. Thus, the b_head variable never changes; it could be replaced with the constant zero.

By now it should be obvious that nothing really gets deleted from the work file. Only the pointers to "deleted" blocks are erased. In order to keep track of all deleted blocks, they are kept on a singly linked *free list*. The global variable b_free points to the first block of the free list. Note that the blocks on the free list are still part of the work file as far as the operating system is concerned.

All changes to the free list are made at its head, as shown in Figures 4a and 4b (page 38). Figure 4a shows the list before a block is deleted while Figure 4b shows the block afterwards. Inserting a block into the free list is essentially the reverse process.

When inserting a block into the work file, a new block is taken from the free list if the free list is not empty. Otherwise, the sysseek() routine is used to allocate a new block from the disk. The variable b_max_diskp keeps track of the last block that has been allocated to the work file. Since blocks are never deallocated, this variable is only incremented, never decremented.

Inserting and Deleting Lines

We have just seen how to insert and delete entire blocks. Let us now take a closer look at the format of each disk block (see Figure 5, page 39). A block consists of an 8-byte header field followed by a data field. The header contains the two link fields, d_next and d_back, which we have already discussed. The header also contains two other fields. The d_lines field is a count of the number of lines contained in the data field. The d_avail field is the number of unused bytes in the block.

Lines are not split across blocks; if a block is not big enough to hold another line, one or more lines are copied to a new block. Almost all blocks contain a few unused bytes, but this is not a great disadvantage. (The average amount of waste in a block is one half of the average size of a line.) The number of lines in the data field is indicated by the d_lines field. Each line is preceded by a two-byte count of the number of characters in the line. No end-of-line character is used in the work file; thus, the net overhead due to the count field is just one byte.

To delete a line the data area is compressed to eliminate the line, the d_lines field is decremented, and the d_avail field is increased by the length of the line. Similarly, to insert a line a hole is made in the data area for the new line, the new line is copied into the hole, and the header (the d_lines and the d_avail fields) is adjusted to reflect the change.

Whenever a line is deleted, a check is made of the neighboring blocks to see whether any blocks can be merged. Merging is very important as it drastically reduces wasted space in the work file. Two blocks are merged by copying the data field of the second block to the available area of the first block. The header of the first block is adjusted and the second block is deleted. This is illustrated in Figures 6a and 6b (page 39).

Things get a bit complicated if there is not enough room in the block for an added line. There are three cases (see Figures 7a, 7b and 7c, page 43), depending on how big the new line is and where in the block the new line goes. Case 1: If the new line will fit in the original block, but the following lines will not, then a new block is created for the following lines. Case 2: If the new line will not fit in the original block, a new block is created for it. If the following lines fit in this new block, they also are copied to the new block. Case 3: If the following lines do not fit in the new block, a *second* new block is created just for them.

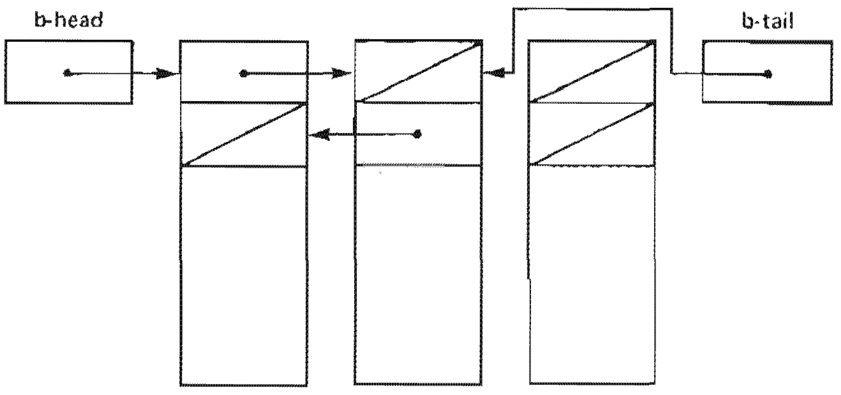
Replacing one line by another is done simply by deleting the old line and inserting the new line. This may seem a bit slow, but the delay is not noticeable. More importantly, using the delete routine in this way insures that blocks will be merged whenever possible. This crude approach to replacing lines saves a lot of code.

Moving through the File

How can the buffer routines find the start of a particular line in the work file? This creates surprising problems. When I first designed the buffer routines, I planned to use an *index*, an array of pointers to the start of each line. I did not want to limit the index to what would fit into memory since the size of the index would then limit the size of the file that could be edited. (Many editors which purport to be able to handle large files actually suffer this limitation including, for example, the editors derived from the book *Software Tools*.) I considered putting at least part of the index on a file. At first this seemed feasible, but in fact it is not. The reason is that the index must be accessed just like an array: given a line, we want to find the pointer to the line *without searching*. But making insertions or deletions to an array requires moving *all* entries in the array which follow the inserted or deleted entry. This is much too expensive if the array is held on a disk file.

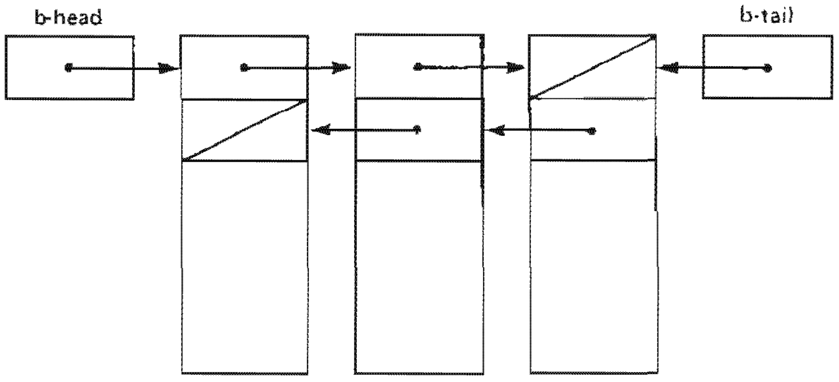
I also investigated using the kinds of data structures common in data base management systems. An example is the B-Tree. (See "The Ubiquitous B-Tree," *ACM Computing Surveys,* Volume 11, Number 2, June 1979.) The basic problem with all these structures is that they assume that the key associated with a data item is fixed. But in the case of an editor, a data item is a line of text and the associated key is the line number. The line number changes whenever an insertion or deletion is made to the work file. In short, I can find no way to adapt B-Trees, or any similar structure, to the problem of creating an index for a text editor. If anyone knows of a way, I would love to hear about it. (Aside: in some editors the line number of a line does *not* change when insertions or deletions are made. Such editors can, and do, use index files.)

With no index available, the buffer



Deleting blocks: after
Inserting blocks: before

Figure 3a.
Inserting or deleting a block at the end of the work file is a special case.



Deleting blocks: before
Inserting blocks: after

Figure 3b.

routines must search through the file block by block. Surprisingly enough, this search is almost always very fast! There are three reasons for this:

First, the requested line is usually very near the current line. In fact, most often the requested line is either the immediately preceding or following line. For example, when scrolling the screen or when using the "find" command, the editor steps through the file line by line.

Second, when the requested line is far away from the current line the new line is most often near either the beginning or end of the file. The beginning and ending blocks of the work file can be accessed without searching because the editor always keeps track of where they are. (This is the reason for the b_max_line and b_tail variables.)

Third, each block contains a count of the number of lines in the block. Thus, it is simple to tell whether a block contains the desired line without actually searching the data field of the block.

Virtual Memory and The LRU Algorithm

Up until now I have been tacitly assuming that all blocks have been loaded into memory when needed. In other words, the main memory appears to be of unlimited size. Since the work file may grow to be much larger than the available main memory, blocks must be swapped into and out of memory as needed. Here is how virtual memory works:

The buffer routines contain a set of *slots*, each of which contains one block and some status information which is not saved on the disk. This status information tells whether the slot is free or occupied, where on the disk the block is stored, and other information which we will discuss a little later.

Blocks are loaded into a slot only by explicitly calling the swap_in() routine. Swap_in() first checks to see whether the requested block already resides in a slot. If so, swap_in() simply returns the slot's number. Otherwise, swap_in() searches for an empty slot. If one is found, the requested block is loaded into the empty slot.

If no empty slot is found, a slot must be swapped out to the disk. The slot which is chosen is the Least Recently Used (LRU) slot. Every time a slot is accessed (i.e., every time swap_in() is called), the d_lru field in the status table is adjusted in such a way that the least recently used slot has the highest value. When a slot must be swapped out, the slot with the largest d_lru field is chosen.

Once the slot has been chosen, the swap_in() routine checks the status table to see if the block has been changed since it was loaded into the slot. This information is popularly called the *dirty bit*, which is part of the d_status field. The dirty bit is cleared when a block is first loaded into a slot, and it is set whenever the block is changed. If the dirty bit is set, the block must be swapped out to the disk before the new block replaces it. Otherwise, the new block can be loaded in over the old block because the disk still contains a correct copy of the old block.

Using a dirty bit complicates the buffer routines quite a bit. For instance, failing to set the dirty bit properly creates a "time bomb." The code appears to work correctly, but the disk contains an incorrect copy of a block. The error only shows up when (possibly much later) the block is reloaded into memory. Tracy Kidder discusses this problem in a different context in his Pulitzer Prize winning book, *The Soul of a New Machine*. (If you haven't yet read it, drop everything and do so at once!) In spite of what I have just said about the dirty bit, using it is *essential*. Without the dirty bit, many more disk accesses will be needed.

In practice, the LRU algorithm works well. It eliminates disk accesses when editing in a localized area, which is most of the time. It also performs reasonably well when scrolling through the file.

It is instructive to compare the performance of the LRU algorithm with a simpler approach. For example, Figure 8 (page 44) shows one of my first ideas for creating a virtual memory. Briefly, a memory buffer contains just a few lines. All lines before these lines are held in a *head file*; all lines after the lines in the buffer are held in a *tail file*. The advantage of this scheme is that all disk activity is to the physical end of each file. This method can be programmed even on systems which do not support random access to files. Also, lines are changed only in the main buffer; there is no need to merge or split blocks.

Using two files in this way, however, is flawed. Moving through the file requires that lines be moved through the memory buffer and written to one of the two files. For instance, moving to the start of the file requires that most of the file be written to the tail file. A little thought should convince you that the performance of this method is essentially the same as using the LRU algorithm *without* the dirty bit.

Performance

N+1 trivial tasks are expected to be accomplished in the same time as N trivial tasks.
— Gray's Law of Programming

N+1 trivial tasks take twice as long as N trivial tasks for N sufficiently large.
— Logg's Rebuttal to Gray's Law

In this section, I'd like to compare the speed of the buffer routines used in ED2 and RED. It is important to realize that you may tune the performance of RED's buffer routines by changing some compile time-constants defined on file red 10.sc (see Listing, page 50). The speed-up may be dramatic, depending on your disk drives.

In general, RED will run faster the more buffers there are and the larger each buffer is. You change the *size* of the buffers by changing the DATA_SIZE constant. You change the *number* of buffers by changing the NSLOTS constant. If you have room in memory, I recommend setting DATA_SIZE to 1024 and NSLOTS to 4 (or more).

Reading a file into the buffer is always considerably slower with these new routines because the file being edited must be copied to the work file. The important fact is that large files are loaded at the same *rate* as small files.

Scrolling the screen will be interrupted from time to time when the LRU algorithm swaps blocks in and out. On a floppy disk this delay is slightly annoying but tolerable. On a hard disk the delay is barely noticeable. In any event, the delay does not depend on the size of the work file, but rather on the speed of the disk drive, the block size, and the position of the work file. (The closer the work file is to the directory, the better.)

Inserting or replacing lines is extremely fast, unless the operation results in a block being swapped in or out. For large files, these operations are *much faster* than with the old buffer routines.

Deleting many lines can take longer with these routines because the deleted blocks must still be written to the disk in order to keep the free list up to date.

Keep in mind that splitting or joining blocks usually does not cause blocks to be swapped from the disk because the LRU algorithm increases the odds that those blocks are already in memory. These operations cause little overhead.

Recovery from Errors

The new buffer routines employ two types of error checking. Whenever a change is made to a block, a check is made of the consistency of the information in the block's header and data fields. These checks are quite effective; almost all bugs in the code were caught this way. The checks have even pointed out a bad memory chip. You can disable these checks by commenting out all the calls to check_block() *on file* red12.sc. I don't recommend that you do this because the checks do not slow down the editor at all. On the other hand, if you eliminate all calls to check_block() you don't have to #include the file red13.sc. This will save you some space if you really need it.

The other type of error checking is reported by the operating system to the sysseek(), sysread(), and syswrite() routines. Recovering from these errors is a difficult problem. Ideally, the editor could keep enough information around to "back out" of any transaction that cannot be completed, but this cannot be done unless a truly enormous amount of information is passed between the routines that detect errors and routines that might do something about the errors. The following paragraphs discuss the three main types of disk error and how RED recovers.

(1) The disk becomes full. This is by far the most likely cause of a disk error. The error is reported on the screen and the operation which caused the error is aborted. The buffer is cleared if the load command caused the error.

(2) The disk is removed during editing. This creates a real problem because CP/M completely botches the error recovery. CP/M gives the message, "BDOS error on A: select," then proceeds to mark the disk as read only! If a write is in progress, CP/M will then abort the editor without ever returning control. So *never* remove a disk while RED is running.

If this kind of error should happen to you, remember that the disk still contains a copy of the work file. To recover the information from the work file do the following:

- Rename the work file so that you don't erase it by invoking the editor!
- Write a short *recovery program*. This program should be patterned after the write_file() routine in file red12.sc. The recovery program should copy the work file to another file deleting header information as it goes. Note that block 0 of the work file *always* contains the first block of the file, but the following blocks may be anywhere in the work file. Use the d_next field in the header of each block to tell where the next block is to be found.

(3) There is a "real" disk error. In other words, the hardware that controls the disk fails to read or to write a sector properly. Some case could be made for retrying the read or write operation a few times to see if the problem would go away. In my experience, this kind of problem is very rare, so the buffer routines don't attempt any fancy recovery. Besides, accessing the disk further might just compound the problem. The best thing to do is to stop editing and start figuring out why the disk error has occurred.

In short, the new buffer routines are probably more reliable than the old routines. Even in an extreme emergency, the work file may be used to recover your work.

In the ED2 editor, errors were indicated by returning a status value from all buffer routines. In RED, this is done by setting three disk restart points – one for each mode that the editor can be in. When a disk error occurs, the longjmp() function (defined in the BDS C library) is used to perform a jump to the current restart point. Here is an example of a *goto* instruction being justified on the basis of eliminating a lot of extraneous code.

Summary

To summarize the important features of RED's new buffer routines:

- **Huge files may be edited** because main memory contains no data structures whose size depends on the size of the file. In addition, the work file never contains much garbage – typically, the work file is no more than 10% larger than the original file being edited. Files are limited in size only by (1) how much room exists on the disk for the work file and (2) how large the operating system allows a file to be.
- **The response time for editing is constant.** Inserting, deleting, or replacing a line takes the same time regardless of the length of the file being edited. I have edited files of 20,000 lines easily.
- **The response time for file operations is linear.** Reading, writing, and appending files proceed at a constant rate, regardless of the size of the files. In other words, reading a 10,000 line program takes only 10 times longer than reading a 1,000 line program.

Next month I'll discuss the Small C library used in RED and improvements that might be made to RED. I'll also say a few words about why RED is copyrighted.

■■J

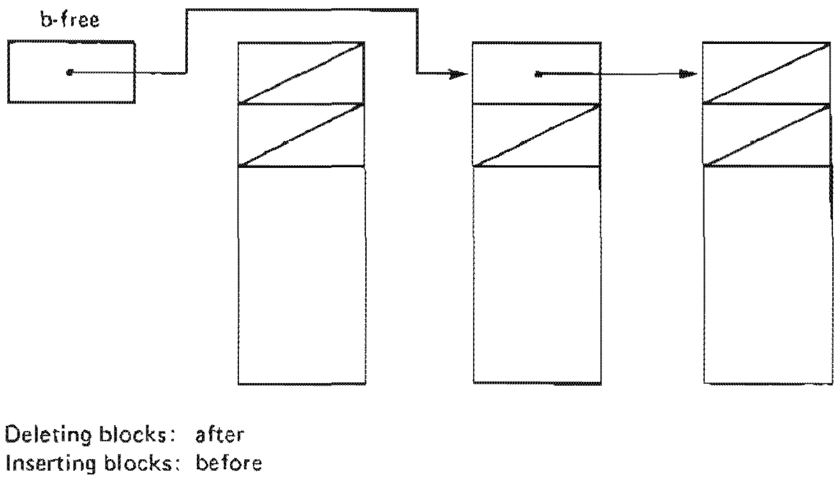


Figure 4a.

All insertions and deletions are made at the head of the free list.

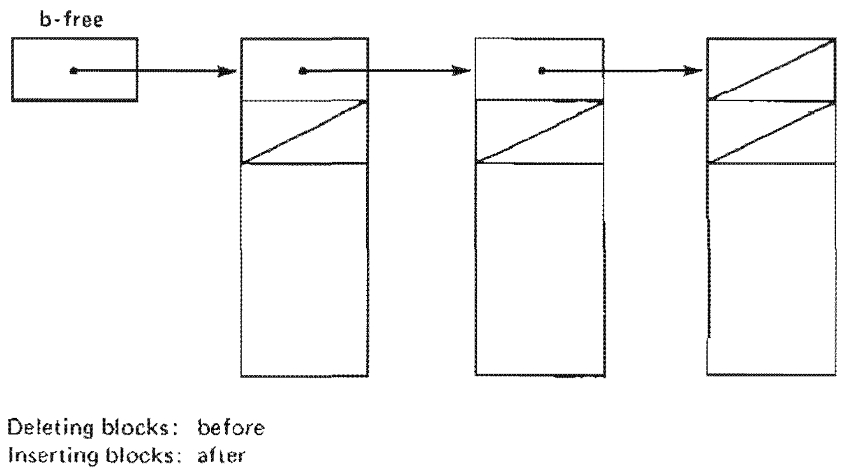


Figure 4b.

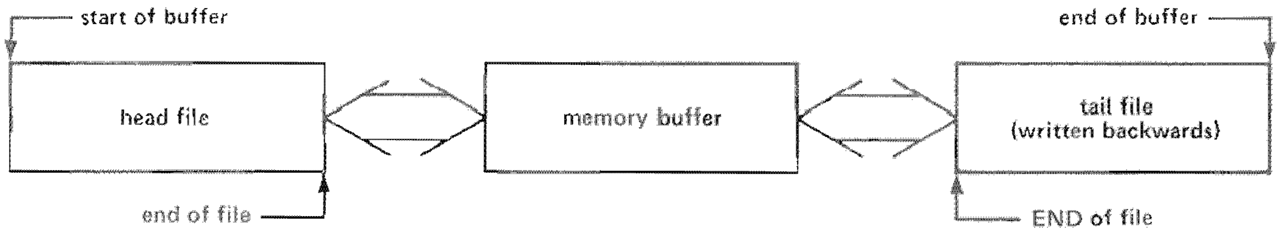


Figure 8.
An alternative way of providing a huge buffer. This method causes many extra disk accesses.

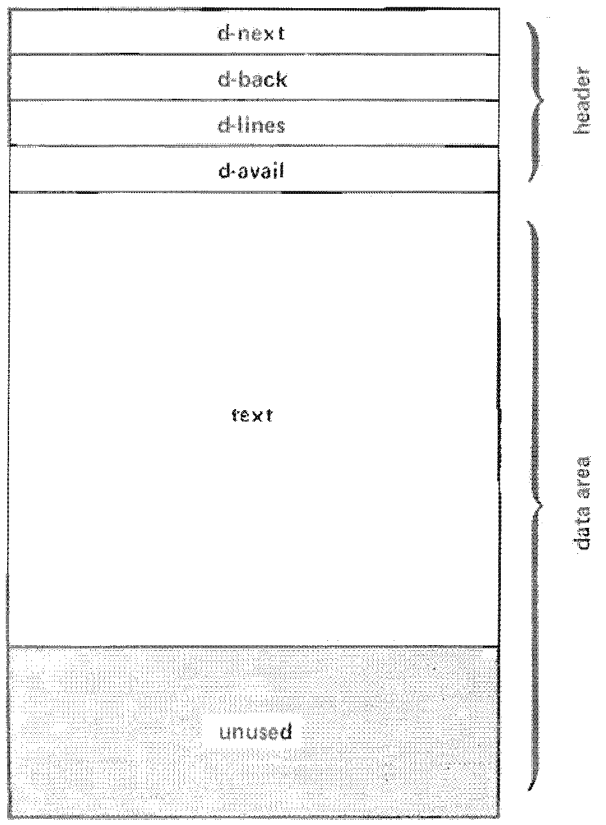


Figure 5.
A block consists of a header field, which tells what is stored in the block, and the data field, which contains lines of text.

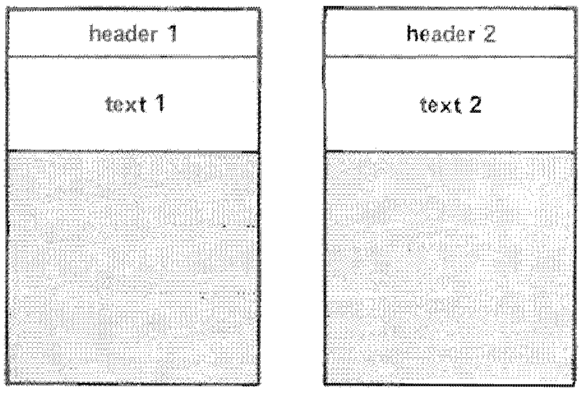


Figure 6a.
Two blocks before they are merged.

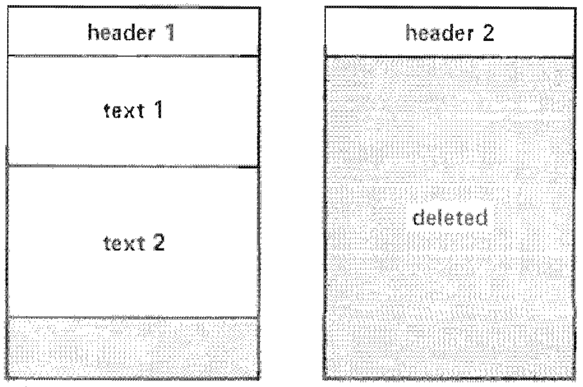


Figure 6b.
Two blocks after merging. Lines are copied from the second block, the header of the first block is adjusted and the second block is deleted.

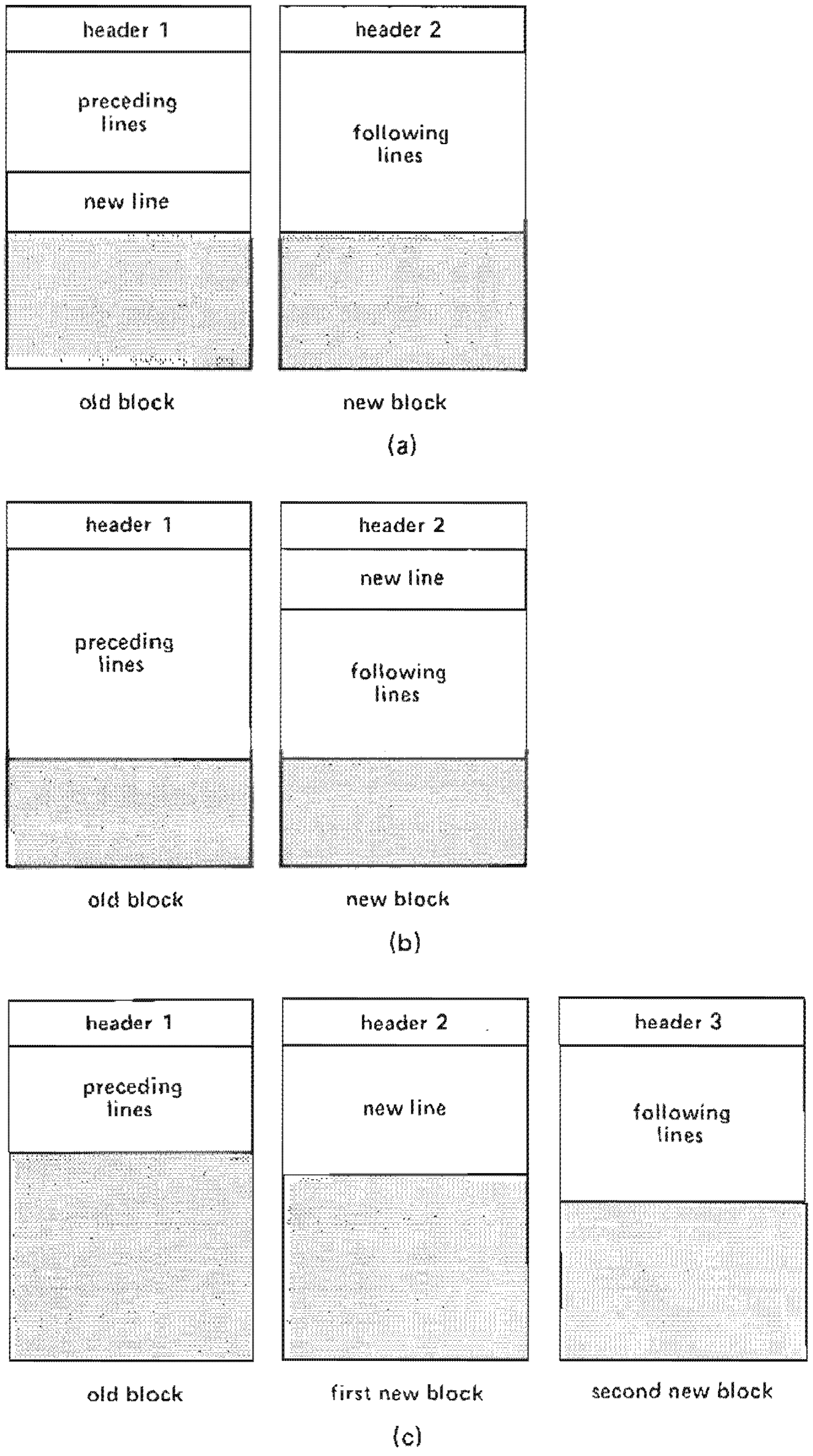


Figure 7.
In (a), the new line fits into the old block; (b) shows both the new line and the following lines fitting into the new block; and (c) shows two new blocks created.

RED - Listing

(Text begins on page 34)

```
/*
        Include file for RED:  small-C version

        Source:  red.sc
        Version: January 20, 1983

        Copyright (C) 1983 by  Edward K. Ream
*/

#include a:red0.sc          /* constants, (system) globals  */
#include a:red1.sc          /* user defined globals         */
#include a:red2.sc          /* main program                 */
#include a:red3.sc          /* command mode commands        */
#include a:red4.sc          /* edit mode module             */
#include a:red5.sc          /* output format module         */
#include a:red6.sc          /* display screen module        */
#include a:red7.sc          /* prompt line module           */
#include a:red8.sc          /* operating system module      */
#include a:red9.sc          /* utility routines             */
#include a:red10.sc         /* buffer module -- part 1      */
#include a:red11.sc         /*               -- part 2      */
#include a:red12.sc         /*               -- part 3      */
#include a:red13.sc         /* debugging routines           */
#include a:lib1             /* machine library              */
#include a:lib2             /* init, exit, utility library  */
#include a:fiolib1          /* file library                 */
#include a:xlib             /* setjmp, longjmp              */
```

```
/*
        RED's non-user defined globals -- small-C version

        Source:  red0.sc
        Version: see below

        Copyright (C) 1983 by Edward K. Ream
*/


/* Define global constants */

#define SIGNON  "Welcome to RED"
#define VERSION "Small-C Version 4.04,  February 27, 1983"
#define COPYRIGHT "Copyright (C) 1983 by Edward K. Ream"
#define XSIGN  "Type 'help' in command mode for help"
#define XSIGN1 "Type 'h' in edit mode for more help"

/* Define constants describing a text line */

#define MAXLEN  200     /* max chars per line           */
#define MAXLEN1 201     /* MAXLEN + 1                   */

/* Define operating system constants */

#define SYSFNMAX 15     /* CP/M file name length + 1    */
#define CPMEOF 26

/* Define misc. constants */

#define EOS     0       /* code sometimes assumes \0    */
#define OK      1
#define ERR     -1      /* error.  must be <0           */
#define ERROR   -1      /* error.  must be <0           */
#define EOF     -2      /* end of file.  must be <0     */
#define YES     1       /* must be nonzero              */
#define NO      0
#define CR      13      /* carriage return              */
#define LF      10      /* line feed                    */
#define TAB     9       /* tab character                */
#define HUGE    32000   /* practical infinity           */
```

```
/*
RED special key definitions
Source:  red1.sc
This file was created by the configuration program:
January 20, 1983.
*/

/*
Define which keys are used for special edit functions.
*/

#define UP1 10
#define DOWN1 13
#define UP2 21
#define DOWN2 4
#define LEFT1 12
#define RIGHT1 18
#define INS1 14
#define EDIT1 5
#define ESC1 27
#define DEL1 8
#define ZAP1 26
#define ABT1 24
#define SPLT1 19
#define JOIN1 16
#define REP1 1

/*
Define length and width of screen and printer.
*/

#define SCRNW 80
#define SCRNW1 79
#define SCRNL 24
#define SCRNL1 23
#define SCRNL2 22
#define LISTW 132
```

```
/*
        RED operating system module -- small-C version

        Source:  red8.sc
        Version: January 23, 1983

        Copyright (C) 1983 by Edward K. Ream
*/


/* Define globals used by this module */

char sysinbuf[128];        /* file buffer */
int  sysincnt;             /* buffer count */

char syscbuf[MAXLEN];      /* console type ahead buffer */
int  syscnt;

int sysrcnt;               /* repeat count */
int syslastc;              /* last character (may be repeated) */

int  systopl,systopy,sysnl;       /* interrupt information */

#define READ_SIZE 4      /* see also definition in red10.sc */



/*      Initialize the system module. */

sysinit()
{
        sysnl    = 0;
        syscnt   = 0;
        sysrcnt  = 0;
        syslastc = 0;
}


/*      Save info for interrupted screen update. */

sysintr(sysl, systy, sysn)
int sysl, systy, sysn;
{
        systopl = sysl;
        systopy = systy;
        sysnl   = max(0,sysn);
}


/*
        Return -1 if no character is ready from the keyboard.
        Otherwise, return the character.

        This routine handles typeahead and the repeat key.
*/

syscstat()
{
        int c, i;

        /* Always look for another character. */
        c = bdos(6,-1);

        if ( (c == REP1) & (syslastc != 0) ) {
                sysrcnt = max(1, 2*sysrcnt);


                i = 0;
                while ( (i++ < sysrcnt) & (syscnt < MAXLEN) ) {
                        syscbuf [syscnt++] = syslastc;
                }
        }
        else if (c != 0) {
                syslastc = c;
                sysrcnt  = 0;
                syscbuf [syscnt++] = c;
        }

        if (syscnt > 0) {
                return syscbuf [--syscnt];
        }
        else {
                return -1;
        }
}
```

(Continued on page 48)

```
/*
        Wait for next character from the console.
        Do not echo it.
        This routine prints any waiting lines if there is no
*/      input ready

syscin()
{
        int c;

        while ((c=syscstat()) == -1) {

                /* Output queued ? */
                if (sysnl > 0) {
                        bufout(systopl, systopy, sysnl);
                }
        }
        return c;
}


/*
        Wait for all console output to be finished.
*/

syswait()
{
        while (sysnl > 0) {
                bufout(systopl, systopy, sysnl);
        }
}


/* Print character on the console. */

syscout(c)
char c;
{
        bdos(6,c);
        return(c);
}


/* Print one character on the printer. */

syslout(c) char c;


{
        bdos(5,c);
        return(c);
}


/* Close a file which was opened by sysopen() or syscreat(). */

sysclose(file)
int file;
{
        return close(file);
}


/*
        Create a file.  Erase it if it exists.
        Leave the file open for read/write access.
*/

syscreat(filename)
char * filename;
{
        return creat(filename);
}


/* Return YES if the file exists. */

sysexists(filename)
char * filename;
{
        int file;

        if ((file = open(filename, 0)) != ERROR) {
                close(file);
                return YES;
        }
        else {
                return NO;
        }
}


/*
        Open a file which already exists.
        Mode 0 -- read only.
        Mode 1 -- write only.
        Mode 2 -- read/write.
*/

sysopen(name, mode)
char *name;
int mode;
{
        /* Kludge:  set count for sysgetc(). */
        if (mode == 0) {
                sysincnt = 128;
        }

        return open(name, mode);
}


/*
        Read next line from a file.

        End the line with a zero byte.
        Only one file at a time may use this routine.
*/

sysrdln(file, buffer, maxlength)
int file;
char *buffer;
int maxlength;
{
        int c, count;

        count = 0;
        while(1) {
                c = sysgetc(file);
                if (c == CR) {
                        continue;
                }
                else if (c == CPMEOF) {
                        buffer [count = EOS];
                        return ERROR;
                }
                else if (c == LF) {
                        buffer [count] = EOS;
                        return count;
                }
                else if (count < maxlength - 1) {
                        buffer [count++] = c;
                }
                else {
                        count++;
                }
        }
}


/*
        Get one character from the input file.
        Only one file at a time may use this routine.
*/


sysgetc(file)
int file;
{
        int n;

        if (sysincnt == 128) {
                n = read(file, sysinbuf, 1);
                if (n == ERROR) {
                        disk_error("read error");
                        return CPMEOF;
                }
                else if (n == 0) {
                        /* End of file. */
                        return CPMEOF;
                }
                else {
                        sysincnt = 0;
                }
        }
        return sysinbuf [sysincnt++];
}


/*
        Read one block (READ_SIZE sectors) into the b[illegible]
*/

sysread(file, buffer)
int file;
char * buffer;
{
        return read(file, buffer, READ_SIZE);
}


/*      Write n sectors from the buffer to the file. */

syswrite(file, buffer, n)
int file;
char * buffer;
int n;
```

(Continued on page 50)

RED - Listing

(Listing continued, text begins on page 34)

```
{
        return write(file, buffer, n);
}


/* Seek to a specified block of an open file. */

sysseek(file, block)
int file, block;
{
        return seek(file, block * READ_SIZE, 0);
}


/* Remove the file from the file system. */

sysunlink(filename)
char * filename;
{
        return unlink(filename);
}


/* Check file name for syntax. */

syschkfn(args) char *args;
{
        return(OK);
}


/* Copy file name from args to buffer. */

syscopfn(args,buffer) char *args, *buffer,
{
int n;
        n=0;
        while (n<(SYSFNMAX-1)) {
                if (args[n]==EOS) {
                        break;
                }
                else {
                        buffer[n]=args[n];
                        n++;
                }
        }
        buffer[n]=EOS;
}
```

```
/*
        RED buffer routines -- small-C version
        Part 1 -- goto, output and status routines.

        Source:  red10.sc
        Version: August 28, 1982; January 23, 1983

        Copyright (C) 1983 by Edward K. Ream
*/


/*
        You may tune these constants for better performance.

        DATA_SIZE:  The size of struct BLOCK.
                    Make sure that DATA_SIZE is a multiple
                    of the size of your disk sectors.
                    (for CP/M, a multiple of 128)
                    Make sure that the READ_SIZE constant in
                    ed0.c is DATA_SIZE / 128.

        NSLOTS:     The number of BLOCKS resident in memory.
                    The code assumes this number is AT LEAST 3.

        DATA_FILE:  The name of the work file. Note the double
                    quotes.  Pick a name you never use.
*/

#define DATA_SIZE 512
#define NSLOTS 5
#define DATA_FILE "@@data@@.tmp"


/* Do not touch this constant. */

#define CPM_SIZE 128


int     b_fatal;        /* erase buffer on disk error   */
int     b_cflag;        /* buffer changed flag          */

int     b_line;         /* current line number          */
int     b_max_line;     /* highest line number          */
char *  b_linep;        /* pointer to line (local var)  */

int     b_slot;         /* current block's slot number  */
int     b_start;        /* first line of current block  */
int     b_head;         /* first block's disk pointer   */
int     b_tail;         /* last block's disk pointer    */
int     b_max_diskp;    /* last sector allocated        */

int     b_data_fd;      /* file descriptor of data file */
int     b_user_fd;      /* file descriptor of user file */
int     b_free;         /* head of list of free blocks  */

char    b_buff [DATA_SIZE];     /* temporary buffer.    */



/*
        Partially define the format of a block.  The data
        field is organized as a singly linked list of lines;
        that is, each line is preceded by a two-byte length
        field.
        The d_back and d_next fields in the header are used
        to doubly-link the disk blocks so that stepping
        through the blocks either forward or backwards is
        efficient.  -1 denotes the end of each list.

        When blocks become totally empty they are entered
        on a list of free blocks.  The links of this list
        are kept in the blocks themselves in the d_next field.
        The b_free variable is the head of this list.

        Also define the in-core block table.  This table
        contains the blocks that have been swapped into
        memory.  Each entry in this table is called a slot.
*/

/* comment out ----------

struct BLOCK {
        int     d_back;
        int     d_next;
        int     d_avail;
        int     d_lines;
        char    d_data [BUFF_SIZE];
}d_blocks [NSLOTS];

---------- end comment out */


#define HEADER_SIZE 8               /* size of first 4 fields  */
#define BUFF_SIZE 504               /* DATA_SIZE - HEADER_SIZE */
#define SLOT_SIZE 2560              /* NSLOTS * DATA_SIZE      */


int  d_back  [NSLOTS];              /* # of previous block  */
int  d_next  [NSLOTS];              /* # of next block      */
int  d_avail [NSLOTS];              /* # of data bytes free */
int  d_lines [NSLOTS];              /* # of lines on block  */
char d_data  [SLOT_SIZE];           /* resident blocks      */




/*
        Define the resident status table.
        There is one entry for each slot.
*/

#define FREE    1       /* status:  block is available  */
#define FULL    2       /* status:  block is allocated  */
#define DIRTY   3       /* status:  must swap out       */

/* comment out ----------

struct STATUS {
        int     d_lru;
        int     d_status;
        int     d_diskp;
} d_stat_tab [NSLOTS];

---------- end comment out */

int     d_lru    [NSLOTS];      /* lru count            */
int     d_status [NSLOTS],      /* FULL, FREE or DIRTY  */
int     d_diskp  [NSLOTS];      /* disk pointer         */


/*


        Boundary conditions:

        1.  Only bufins() can extend the buffer, NOT
            bufgo() and bufdn().

        2.  bufatbot() is true when the current line is
            PASSED the last line of the buffer.  Both
            bufgo() and bufdn() can cause bufatbot() to
            become true.  bufgetln() returns a zero length
            line if bufatbot() is true.

        3.  b_max_line is the number of lines in the buffer.
            However, b_line == b_max_line + 1 is valid and
            it means that b_line points at a null line.
        4.  All buffer routines assume that the variables
            b_slot, b_line and b_start describe the
            current line when the routine is called.  Thus,
            any routine which changes the current line must
            update these variables.
*/


/* Return YES if at bottom of buffer (past the last line). */

bufatbot()
{
        return (b_line > b_max_line);
}


/* Return YES if at top of buffer  */

bufattop()
{
        return (b_line == 1);
}


/* Return YES if the buffer has been changed. */

bufchng()
{
        return b_cflag;
}


/* Move towards end of buffer. */

bufdn()
{
        /* The call to buf_gofw() instead of bufgo()
         * is made purely to increase speed slightly.
         */
        if (bufatbot()) {
                return;
        }
        else {
                b_line++;
                buf_gofw();
        }
}


/* Clean up any temporary files. */

bufend()
{
        sysunlink(DATA_FILE);
}


/*
        Go to line n.
        Set b_slot, b_line, b_start.
*/

bufgo(n)
int n;
{
        int distance, oldline;

        /* Put the request in range. */
        oldline = b_line;
        b_line = min (n, b_max_line + 1);
        b_line = max (1, b_line);
        distance = b_line - oldline;

        if (distance == 0) {

                /* We are already at the requested line. */
                return;

        }
        else if (distance == 1) {

                /* Go forward from here. */
                buf_gofw();
                return;
        }
        else if (distance == -1) {

                /* Go back from here. */
                buf_gobk();
                return;

        }
        else if (distance > 0) {
                if ( b_line >
                     oldline + ((b_max_line - oldline) / 2)
                   ) {

                        /* Search back from end of file. */
                        swap_in(b_tail, &b_slot);
                        b_start =
                            1 + b_max_line - d_lines [b_slot];
                        buf_gobk();
                        return;
                }
                else {

                        /* Search forward from here. */
                        buf_gofw();
                        return;
                }
        }
        else {
                if (b_line < oldline / 2) {

                        /* Search from start of file. */
                        swap_in(b_head, &b_slot);
                        b_start = 1;
                        buf_gofw();
                        return;

                }
                else {

                        /* Search back from here. */
                        buf_gobk();
                        return;
                }
        }
}


/*
        Search backwards from block for b_line.
        The starting line number of the block is b_start.
        Set b_slot and b_start.
 */

buf_gobk ()
{
        int diskp;

        if ( (b_slot == ERROR) |
             (b_start < 1)      | (b_start > b_max_line) |
             (b_line  < 1)      | (b_line  > b_max_line + 1)
           ) {

                cant_happen("buf_gobk 1");
        }

        /* Scan backward for the proper block. */
        while (b_start > b_line) {

                /* Get the previous block in memory. */
                diskp = d_back [b_slot];
                if (diskp == ERROR) {
                        cant_happen("buf_gobk 2").
                }
                swap_in(diskp, &b_slot);

                /* Calculate the start of the next block. */
                b_start = b_start - d_lines [b_slot];
                if (b_start <= 0) {
                        cant_happen("buf_gobk 3");
                }
        }
}


/*
        Search forward for line n.
        Set b_slot and b_start.
 */

buf_gofw ()
{
        int diskp;

        /* The last line is always null. */
        if (bufatbot()) {
                return;
        }

        if ( (b_slot == ERROR) | (b_start < b_start) |
             (b_start < 1)      | (b_start > b_max_line) |
             (b_line  < 1)      | (b_line  > b_max_line + 1)
           ) {

                cant_happen("buf_gofw 1");
        }

        /* Scan forward to the proper block. */
        while (b_start + d_lines [b_slot] <= b_line) {


                /* Get the start of the next block. */
                b_start = b_start + d_lines [b_slot];

                /* Swap in the next block. */
                diskp = d_next [b_slot];
                if ((diskp == ERROR) | (b_start > b_max_line)){
                        cant_happen("buf_gofw 2");
                }
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 34)

```
                swap_in(diskp, &b_slot);
        }
}


/* Return the current line number. */

bufln()
{
        return b_line;
}


/* Initialize the buffer module. */

bufnew()
{
        int i;
        char * static,

        /*
                Initialize b_data_fd on the first call
                to this routine.  A kludge is required
                because small-C has neither real static
                variables nor initializers.
        */


        static = "";
        if (*static == 0) {
                *static = 1;
                b_data_fd = ERROR;
        }

        /* The free list is empty. */
        b_free = ERROR;

        /* Free all slots. */
        i = 0;

        while (i < NSLOTS) {
                d_status [i] = FREE;
                d_lru    [i] = i;
                d_diskp  [i] = ERROR;
                i++;
        }
        /* Allocate the first slot. */
        b_slot = b_head = b_tail = 0;
        b_max_disp = 1;
        d_diskp  [b_slot] = 0;
        d_status [b_slot] = DIRTY;
        d_back   [b_slot] = d next [b_slot] = ERROR;

        /* The first slot is empty. */
        d_lines [b_slot] = 0;
        d_avail [b_slot] = BUFF_SIZE;

        /* Make sure temp file is erased. */
        if (b_data_fd != ERROR) {
                sysclose(b_data_fd);
                b_data_fd = ERROR;
                sysunlink(DATA_FILE);
        }

        /* Set the current and last line counts. */
        b_line = 1;
        b_max_line = 0;
        b_start = 1;

        /* Indicate that the buffer has not been changed. */
        b_cflag = NO;

        /* Do not erase the work file on a disk error. */
        b_fatal = NO;
}


/* Return YES if buffer is near the bottom line */

bufnrbot()
{
        return (b_line >= b_max_line);
}

/* Put nlines lines from buffer starting with line topline at
 * position topy of the screen.
 */

        [illegible], topy, nlines)
        [illegible]        nlines;


        [illegible] x, y;

        [illegible]x();
        y = outgety();
        l = b_line;
        while (nlines > 0) {
                outxy(0, topy++);
                bufoutln(topline++);
                nlines--;
                sysintr(topline, topy, nlines);
                break;
        }
        outxy(x,y);
        bufgo(l);
}


/* Print one line on screen. */

bufoutln(line)
int line;
{
        char buffer [MAXLEN1],
        int n;

        bufgo(line);
        if (bufatbot()) {
                outdeol();
        }
        else {
                n = bufgetln(buffer, MAXLEN),
                n = min(n, MAXLEN);
                buffer [n] = CR;
                fmtsout(buffer, 0);
                outdeol();
        }
}



/* Replace current line with the line that p points to
 * The new line is of length n.
 */

bufrepl(line, n)
char line [];
int n;
{
        /* Do not replace null line.  Just insert. */
        if (bufatbot()) {
                bufins(line, n);
                return;
        }
        bufdel();
        bufins(line, n);
}


/* Indicate that the file has been saved. */

bufsaved()
{
        b_cflag = NO;
}


/* Move towards the head of the file. */

bufup()
{
        /* The call to buf_gobk() instead of buf_go()
         * is made purely to increase speed slightly.
         */

        if (bufattop()) {
                return,
        }
        else {
                b_line--;
                buf_gobk();
        }
}
```

```
/*
        RED buffer routines -- small-C version
        Part 2 -- line routines.

        Source:  red11.sc
        Version: August 7, 1982; February 26, 1983

        Copyright (C) 1983 by Edward K. Ream
*/


/*
        Scan for the start of the current line.
        Return *count  = the # of characters in the line.
        Return *prefix = the # of characters before the line.
*/

b_scan(pointer, count, prefix)
int     *pointer;       /* Kludge -- should be char ** */
int     *count;
int     *prefix;
{
        char    *cp;
        int     i, limit, count1;
        /* The last line is always null. */
        if (bufatbot()) {
                *prefix  = BUFF_SIZE - d_avail [b_slot];
                *pointer = data_addr(b_slot) + *prefix;
                *count   = 0;
                return;
        }

        /* Limit is the starting line # of the next block. */
        limit = b_start + d_lines [b_slot];

        /* Point pointer at the start of the first line. */
        cp = data_addr (b_slot);

        /* Keep track of characters before the line. */
        *prefix = 0;

        i = b_start;
        while (1) {

                if (i == limit) {
                        cant_happen("b_scan 1");
                }

                /* Get length of the line. */
                count1    = b_getnum(cp);

                if ( (count1 < 0) | (*prefix >= BUFF_SIZE) ) {
                        cant_happen("b_scan 2");
                }

                /* At the requested line? */
                if (i == b_line) {
                        break;
                }

                /* Step over the count and the line. */
                cp       = cp + count1 + 2;
                *prefix = *prefix + count1 + 2;
                i++;
        }

        /* Point past the line length. */
        cp = cp + 2;

        /* Set values in the calling routines. */
        *pointer = cp;
        *count = count1;
}

/*
        Get and put a 2-byte line number.
        These are machine independent substitutes for casts:
*/

b_getnum(cp)
char *cp;
{
        /*      simulate:  ip = (*int) cp; return *ip.   */

        return (*cp << 8) | *(cp+1);
}

b_putnum(cp, num)
char *cp;
int num;
{
        /*      simulate:  ip = (*int) cp; *ip = num.    */

        *cp      = (num & (255 << 8)) >> 8;
        *(cp+1)  = num & 255;
}


/* Delete the current line. */

bufdel()
{
        char    *p, *p1, *q;
        int     length, junk;
        int     endp;
        int     back, current, next;

        /* Do nothing if the buffer is empty. */
        if (bufatbot()) {
                return;
        }

        /* The current block will become dirty. */
        is_dirty(b_slot);
        b_cflag = YES;

        /*
                Point p  at the line # of the deleted line.
                Point p1 at the line # of the following line.
                Point q  passed the last byte of the block.
        */

        b_scan(&p, &length, &junk);
        endp = BUFF_SIZE - d_avail [b_slot].
        q   = data_addr (b_slot) + endp;
        p1 = p + length;
        p  = p - 2;

        /* Compress the block  */
        while (p1 != q) {
                *p++ = *p1++;
        }

        /* Adjust the avail and line counts in the block. */
        d_avail [b_slot] = d_avail [b_slot] + length + 2;
        d_lines [b_slot]--;

        /* Decrease the overall line count. */
        b_max_line--;

        /* Point to the previous, current and next blocks  */
        back     = d_back  [b_slot];
        current = d_diskp [b_slot];
        next     = d_next  [b_slot];

        /* Move to the correct block. */

        if ( (next == ERROR) & (d_lines [b_slot] == 0) ) {
                /* The last block is empty.  Move back. */
                if (back != ERROR) {
                        swap_in(back, &b_slot);
                        b_start = b_max_line -
                                        d_lines [b_slot] + 1;
                }
        }

        else if (b_start + d_lines [b_slot] == b_line) {
                /* The line moves to the next block. */
                if (next != ERROR) {
                        swap_in(next, &b_slot);
                        b_start = b_line;
                }
        }

        /*
                Combine blocks if possible.
                This is tricky code because combine() causes
                side effects.  Do not try to pre-compute the
                arguments for the second call to combine().
        */

        combine(d_back  [b_slot], d_diskp [b_slot]).
        combine(d_diskp [b_slot], d_next  [b_slot]);


        /* ----- check code ----- */
        check_block("delete");
}


/* Delete n lines starting at the current line. */

bufdeln(n)
int n;
{
        int i;

        i = 0;
        while( (i < n) & (bufatbot() == NO) ) {
                bufdel();
                i++;
        }
}

/*
        Copy the current line from the buffer to line [].
        The size of line [] is linelen.
        Return k = the length of the line.
        If k > linelen then truncate k - linelen characters.
 */

bufgetln(line, linelen)
char    *line;
int     linelen;
{
        int     count, i, junk, limit;
        char    *cp;

        /* Return null line at the bottom of the buffer. */
        if (bufatbot()) {
                line [0] = CR.
                return 0;
        }

        /* Scan forward to start of the line. */
        b_scan(&cp, &count, &junk);

        /* Copy line to buffer */
        limit = min(count, linelen).
        i = 0;
        while (i < limit) {
                line [i] = cp [i];
                i++;
        }

        /* End with zero. */
        line [min (count, linelen - 1)] = EOS;
```

(Continued on next page)

(Listing continued, text begins on page 34)

```
        /* Return the number of characters in the line. */
        return count;
}


/*
        Insert line before the current line.  Thus, the line
        number of the current line does not change.  The line
        ends with a zero byte but not with a CR.

        This is fairly crude code, as it can end up splitting
        the current block into up to three blocks.  However,
        the combine() routine does an effective job of keeping
        the size of the average block big enough.
*/

bufins(line, linelen)
char line [];
int linelen;
{
        char    *p, *q;
        int     junk;
        int     length1;     /* # of chars before break point */
        int     length2;     /* # of chars after  break point */
        int     i;

        if (linelen > BUFF_SIZE) {
                cant_happen("bufins 1");
        }

        /* Point p at the start of the current line. */
        b_scan(&p, &junk, &length1);

        p = p - 2;

        /* Calculate # of characters after break point. */
        length2 = BUFF_SIZE - d_avail [b_slot] - length1;

        /* The current slot is now dirty. */
        is_dirty(b_slot);

        /* Allow for 2-byte line length. */
        if (length1 + linelen + 2 > BUFF_SIZE) {

                split_block(length1, length2, YES);
                length1 = 0;
                length2 = BUFF_SIZE - d_avail [b_slot];
        }

        /*
                At this point we know that the new line will
                fit on the current block at position length1.
        */

        if (linelen + 2 > d_avail [b_slot]) {

                split_block(length1, length2, NO);

                /* Copy line to the end of the old block. */
                p = data_addr (b_slot) + length1;
                i = 0;
                while (i < linelen) {
                        p [2 + i] = line [i];
                        i++;
                }

                /* Insert line length. */
                b_putnum(p, linelen);

                /* Adjust header. */
                d_avail[b_slot] = d_avail[b_slot]-linelen-2;
                d_lines[b_slot]++;
        }
        else {
                /* Make a hole in the block. */
                p = data_addr (b_slot) + length1;
                q = p + linelen + 2;

                i = length2 - 1;
                while (i >= 0) {
                        q [i] = p [i];
                        i--;
                }

                /* Put the line length in the hole. */
                b_putnum(p, linelen);

                /* Copy the new line into the hole  */
                p = p + 2;
                i = 0;
                while (i < linelen) {
                 p [i] = line [i];
                 i++;
         }
         /* Adjust the header. */
         d_avail[b_slot] = d_avail[b_slot]-linelen-2;
         d_lines[b_slot]++;
 }
        /*
                Special case: inserting a null line at the
                end of the file is not a significant change.
         */

        if ( (linelen != 0) | (bufatbot() == 0) ) {
                b_cflag = YES;
        }

        /* Bump the number of the last line. */
        b_max_line++;

        /* ----- check code ----- */
        check_block("bufins");
}


/*
        Combine two blocks into one if possible.
        Make the new block the current block.
 */

combine(diskp1, diskp2)
int diskp1, diskp2;
{
        char                *p1,    *p2;
        int                 slot1,  slot2,  slot3;
        int                 len1,   len2;
        int                 i;

        /* Make sure the call makes sense. */
        if ( (diskp1 == ERROR) | (diskp2 == ERROR) ) {
                return;
        }

        /* Get the two blocks. */
        swap_in(diskp1, &slot1);
        swap_in(diskp2, &slot2);

        if ( (d_next [slot1] != diskp2) |
             (d_back [slot2] != diskp1)
           ) {
                cant_happen("combine 1");
        }

        /* Do nothing if the blocks are too large. */
        len1 = BUFF_SIZE - d_avail [slot1];
        len2 = BUFF_SIZE - d_avail [slot2];

        if ( (len1 > BUFF_SIZE) | (len2 > BUFF_SIZE) ) {
                cant_happen("combine 2");
        }

        if (len1 + len2 > BUFF_SIZE) {
                return;
        }

        /* Copy buffer 2 to end of buffer 1. */
        p1 = data_addr (slot1) + len1;
        p2 = data_addr (slot2);

        i = 0;
        while (i < len2) {
                p1 [i] = p2 [i];
                i++;


        }

        /* Both blocks are now dirty  */
        is_dirty(slot1);
        is_dirty(slot2);

        /* Adjust the back pointer of the next block. */
        if (d next [slot2] != ERROR) {
                swap_in(d_next [slot2], &slot3);
                d_back [slot3] = d_diskp [slot1];
                is_dirty(slot3);
        }

        /*
                Adjust the current block if needed.
                The value of b_start must be decremented
                by the OLD value of d_lines [slot1].
         */

        if (b_slot == slot2) {
                b_slot  = slot1;
                b_start = b_start - d_lines [slot1];
        }

       /* Adjust the header for block 1. */
       d_lines [slot1] = d_lines [slot1] + d_lines [slot2];
       d_avail [slot1] = BUFF_SIZE - len1 - len2;
       d_next  [slot1] = d_next [slot2];

       /* Adjust the pointers to the last block. */
       if (diskp2 == b_tail) {
               b_tail = diskp1;
       }

       /* Slot 2 must remain in core until this point. */
       free_block(slot2);
```

```
        /* ----- check code ----- */
        check_block("combine");
}


/* Put the block in the slot on the free list. */

free_block(slot)
int slot;
{
        /* Link the block into the free list. */
        d_next [slot] = b_free;
        b_free = d_diskp [slot];

        /* Erase the block. */
        d_lines [slot] = 0;
        d_avail [slot] = BUFF_SIZE;
        is_dirty(slot);
}


/*
        Create a new block linked after the current block.
        Return the slot number and a pointer to the new block
*/

/* int */
new_block (slotp)
int     *slotp;


{
        int     slot1,  slot2;
        int     diskp;

        /* Get a free disk sector. */
        if (b_free != ERROR) {

                /* Take the first block on the free list. */
                diskp = b_free;

                /* Put the block in a free slot. */
                swap_in(diskp, &slot1);

                /* Adjust the head of the free list. */
                b_free = d_next [slot1];
        }
        else {
                /* Get a free slot. */
                diskp = ++b_max_diskp;
                swap_new(diskp, &slot1);
        }


        /* Link the new block after the current block. */
        d_next [slot1] = d_next  [b_slot];
        d_back [slot1] = d_diskp [b_slot];
        d_next [b_slot] = diskp;
        if (d_next [slot1] != ERROR) {
                swap_in(d_next [slot1], &slot2);
                d_back [slot2] = diskp;
                is_dirty(slot2);
        }

        /* The block is empty. */
        d_lines [slot1] = 0;
        d_avail [slot1] = BUFF_SIZE;
        is_dirty(slot1);

        /* Set the user's field. */
        *slotp  = slot1;
}


/*
        Split the current block in two pieces.
        Length1 is the number of chars before the break point.
        Length2 is the number of chars after  the break point.
        If flag == YES, make the new block the current block.
*/

split_block(length1, length2, flag)
int length1, length2, flag;
{
        char            *p, *q;
        int             slot2;
        int             i;


         /* Create a new block. */
         new_block(&slot2);

         /* Mark both blocks as dirty  */


         is_dirty(b_slot);
         is_dirty(slot2);

         /* Copy end of the old block to start of the new. */
         p = data_addr (b_slot) + length1;
         q = data_addr (slot2);
```

```
        i = 0;
        while (i < length2) {
                q [i] = p [i];
                i++;
        }

        /* Adjust the headers. */
        d_lines [slot2] = d_lines [b_slot] - (b_line-b_start);
        d_avail [slot2] = BUFF_SIZE - length2;
        d_lines [b_slot] = b_line - b_start;
        d_avail [b_slot] = BUFF_SIZE - length1;

        /* Adjust the pointer to the last block. */
        if (d_diskp [b_slot] == b_tail) {
                b_tail = d_diskp [slot2];
        }

        if (flag == YES) {

                /* Make the new block the current block. */
                b_start = b_start + d_lines [b_slot];
                b_slot  = slot2;
        }
}
```

```
/*
        RED buffer routines -- small-C version
        Part 3 -- file routines.

        Source:  red12.sc
        Version: August 29, 1982;  February 27, 1983

        Copyright (C) 1983 by Edward K. Ream
*/


/* Return the address of the data area for the slot. */

/* char * */
data_addr(slot)
int slot;
{
        return d_data + (slot * DATA_SIZE) + HEADER_SIZE;
}


/* Open the data file. */

/* int */
data_open()
{
        int fd;

        /* Erase the data file if it exists. */
        sysunlink(DATA_FILE);

        /* Create the data file. */
        b_data_fd = syscreat(DATA_FILE);
        if (b_data_fd == ERROR) {
                disk_error("can not open swap file.");
        }

        /* Close the file, reopen it for read/write access. */
        sysclose(b_data_fd);
        b_data_fd = sysopen(DATA_FILE, 2);
        return b_data_fd;
}


/* Make the slot the MOST recently used slot. */

do_lru(slot)
int slot;
{
        int i, lru;

        /*
                Change the relative ordering of all slots
                which have changed more recently than slot
         */
        lru = d_lru [slot];
        i = 0;
        while (i < NSLOTS) {
                if (d_lru [i] < lru) {
                        d_lru [i]++;
                }
                i++;
        }


        /* The slot is the most recently used. */
        d_lru [slot] = 0;
}


/*
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 34)

```
        Print an error message and abort.
        It would be unwise to try to recover from here
        because the state of the work file is unknown.
        However, the work file is left intact so that it
        may be examined and the original file reconstituted.
*/

disk_error(message)
char *message;
{
        error(message);

        /* Erase the buffer for fatal errors. */
        if (b_fatal == YES) {
                bufnew();
        }

        /* Jump to the error recovery point. */
        longjmp(D_ERROR, ERROR);
}


/* Indicate that a slot must be saved on the disk. */

is_dirty(slot)
int slot;
{
        d_status [slot] = DIRTY;
}


/* Put out the block-sized buffer to the disk sector. */

put_block(buffer, sector)
char *buffer;
int sector;
{
        int s;

        /* Seek to the correct sector of the data file. */
        s = sysseek(b_data_fd, sector);
        if (s == -1) {
                disk_error("seek failed in put_block.");
        }

        /* Write the block to the data file. */
        if (syswrite( b_data_fd, buffer, READ_SIZE)
            != READ_SIZE) {
                disk_error("write failed in put_block.");
        }
}


/*
        Fill in the header fields of the output buffer and
        write it to the disk.
*/

/* char * */
put_buf(n)
int n;
{
        int     *p;

        if (n == 0) {
                cant_happen("put_buf 1");
        }

        /*
                Fill in the back and next links immediately.
                This can be done because we are not waiting
                for the LRU algorithm to allocated disk blocks.
                The last block that put_buf() writes will have
                an incorrect next link.  Read_file() will make
                the correction.

                The d_avail field calculation allows for the
                line length in the line just built.
        */
         p = b_buff;
         *p++ = b_max_diskp - 1;             /* d_back field  */
         *p++ = b_max_diskp + 1;             /* d_next field  */
         *p++ = BUFF_SIZE - n;               /* d_avail field */
         *p++ = b_line - b_start;            /* d_lines field */

        /* Update block and line counts. */
        b_max_diskp++;
        b_start = b_line;

        /* Write the block. */
        put_block(b_buff, b_max_diskp - 1);
}


/*
        Write out the slot to the data file.
 */
/* int */
put_slot(slot)
int slot;
{
        int     *p;

        if (d_diskp [slot] == ERROR) {
                cant_happen("put_slot");
        }

        /* Copy header information back to the block. */
        p     = d_data + (slot * DATA_SIZE);
        *p++  = d_back  [slot];
        *p++  = d_next  [slot];
        *p++  = d_avail [slot];
        *p++  = d_lines [slot];

        /* Write the block to the disk. */
        put_block(d_data + (slot * DATA_SIZE), d_diskp [slot]);
}

/* Read a file into the buffer. */

buf_r_file(file_name)
char file_name [];
{
int     i, j;
char    *outbuf;                /* the output buffer     */
int     in;                     /* input buffer index    */
int     out;                    /* output buffer index   */
int     out_save;               /* line starts here      */
int     count;                  /* chars in line         */
int     c;                      /* current char          */
int     *ip;                    /* integer pointer       */

/* Clear the swapping buffers and the files. */
bufnew();
d_status [b_slot] = FREE;

/* Open the user file for reading only. */
b_user_fd = sysopen(file_name, 0);
if (b_user_fd == ERROR) {
        disk_error("file not found");
}

/* Open the data file. */
data_open();

/* Erase the buffer on a disk error. */
b_fatal = YES;

/* The file starts with line 1. */
b_line = 1;
b_start = 1;

/* There are no blocks in the file yet. */
b_head = b_tail = ERROR;
b_max_diskp = 0;

in = DATA_SIZE;                 /* Force an initial read. */
out = 2;
out_save = 0;
b_line = b_start = 1;
outbuf = b_buff + HEADER_SIZE;
count = 0;

while (1) {

        if ((out >= BUFF_SIZE) & (out_save == 0)) {

                /* The line is too long. */
                error ("line split");

                /* End the line. */
                b_putnum(outbuf, count);
                b_line++;
                count = 0;
                        /* Clear the output buffer. */
                        put_buf(out);
                        out = 2;
                        out_save = 0;
                }

                else if (out >= BUFF_SIZE) {

                        /* Write out the buffer. */
                        put_buf(out_save);

                        /* Move the remainder to the front. */
                        i = 2;
                        j = out_save + 2;

                        while (j < out) {
                                outbuf [i++] = outbuf [j++];
                        }

                        /* Reset restart point. */
                        count = out - out_save - 2;
                        out_save = 0;
                        out = count + 2;
                }

                c = read1(&in);
                if (c == CPMEOF) {

                        if (count != 0) {

                            /* Finish the last line. */
                            b_putnum(outbuf + out_save, count);
                            b_line++;
                            out_save = out;
                        }

                        if (out_save != 0) {
                                put_buf(out_save);
                        }
                        break;
                }

                else if (c == LF) {

                        /* Ignore LF's */
                        continue;
                }

                else if (c == CR) {

                        /* Finish the line. */
                        b_putnum(outbuf + out_save, count);

                        /* Set restart point. */
                        b_line++;
                        out_save = out;
                        out = out + 2;
                        count = 0;
                }

                else {

                        /* Copy normal character. */
                        outbuf [out++] = c;
                        count++;
                }
        }

        /* Close the user' file  */
        sysclose(b_user_fd);

        /* Special case:  null file */
        if (b_max_diskp == 0) {
                bufnew();
                return;
        }

        /* Rewrite the last block with correct next field. */
        ip = b_buff;
        ip++;

        *ip = ERROR;
        put_block(b_buff, b_max_diskp - 1).

        /* Set the pointers to the first and last blocks. */
        b_max_diskp--;
        b_head = 0;
        b_tail = b_max_diskp;

        /* Move to the start of the file  */
        b_max_line = b_line - 1;
        b_line = 1;
        b_start = 1;
        swap_in(b_head, &b_slot);
        b_fatal = NO;
}


/* Get one character from the input file. */

read1(in)
int *in;
{

        char    *  c;
        int        s,

        /* Put the input buffer in the first slot. */
        inbuf = d_data;

        if (*in == DATA_SIZE) {

                /* Read the next sector. */
                s = sysread (b_user_fd, inbuf);
                if (s == ERROR) {
                        disk_error("read failed");
                }

                /* Force a CPM end of file mark. */
                if (s < READ_SIZE) {
                        inbuf [s * CPM_SIZE] = CPMEOF;
                }

                *in = 0;
        }

        /* Return the next character of the buffer. */
        return inbuf [(*in)++];
}

/*
```

```
        Put the block from the disk into a slot in memory.
        Return the slot number and a pointer to the block.
 */

/* int */
swap_in(diskp, slotp)
int diskp;
int *slotp;
{
        int     slot, s;
        int     *p;

        if ( (diskp < 0) | (diskp > b_max_diskp) ) {
                cant_happen("swap_in 1");
        }

        /* See whether the block is already in a slot  */
        slot = 0;
        while (slot < NSLOTS) {
                if ( (d_status [slot] != FREE) &
                     (d_diskp  [slot] == diskp)
                   ) {

                        /* Reference the block. */
                        do_lru(slot);

                        /* Set the caller's field. */
                        *slotp  = slot;
                        return;
                }
                slot++;
        }

        /* Clear a slot for the block. */
        swap_new(diskp, slotp);

        /* Seek to the proper place. */
        s = sysseek(b_data_fd, diskp).
        if (s == -1) {
                disk_error("swap_in:  disk full");
        }

        /* Read the block into the slot. */
        s = sysread(b_data_fd, d_data + (*slotp * DATA_SIZE));
        if (s == ERROR) {
                disk_error("swap_in:  disk not ready");
        }

        /* Copy information to arrays for easy access. */
        p = d_data + (*slotp * DATA_SIZE);
        d_back  [*slotp] = *p++;
        d_next  [*slotp] = *p++;
        d_avail [*slotp] = *p++;
        d_lines [*slotp] = *p++;

        /* Swap_new() has already called do_lru(). */
}
/*
        Free a slot for a block located at diskp.
        Swap out the least recently used block if required.
        Return slotp.
 */

/* int */
swap_new (diskp, slotp)
int diskp;
int *slotp;

 {
        int slot;

        /* Search for an available slot. */
        slot = 0;
        while (slot < NSLOTS) {
                if (d_status [slot] == FREE) {
                        break;
                }
                slot++.
        }

        /* Swap out a block if all blocks are full. */
        if (slot == NSLOTS) {
                slot = swap_out();
        }

        /* Make sure the block will be written. */
        d_status [slot] = FULL;
        d_diskp  [slot] = diskp;

        /* Reference the slot. */
        do_lru(slot);

        /* Return slotp. */
        *slotp = slot;
}
/*
        Swap out the least recently used (LRU) slot.
        Return the index of the slot that becomes free.
 */

/* int */
swap_out()
{
        int slot;
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 34)

```
        /* Open the temp file if it has not been opened. */
        if (b_data_fd == ERROR) {
                b_data_fd = data_open();
        }

        /* Find the least recently used slot. */
        slot = 0;
        while (d_lru [slot] != NSLOTS - 1) {
                slot++;
        }

        /* Do the actual swapping out if memory is dirty. */
        if (d_status [slot] == DIRTY) {
                put_slot(slot);
                return slot;
        }

        /* d_diskp is not ERROR if status is not DIRTY. */
        if (d_diskp [slot] == ERROR) {
                cant_happen("swap_out");
        }

        /* Indicate that the slot is available. */
        d_status [slot] = FREE;
        d_diskp  [slot] = ERROR;

        /* Return the slot number. */
        return slot;
}

/* Write the entire buffer to file. */

buf_wfile(file_name)
char *file_name;

        char *data;
        int out, slot, lines, length, next, count;
        int c;

        /* Open the user file.  Erase it if it exists. */
        b_user_fd = syscreat(file_name);
        if (b_user_fd == ERROR) {
                disk_error("create failed");
        }

        /* Copy each block of the file. */
        out = 0;
        next = b_head;
        while (next != ERROR) {

                /* Swap in the next block. */
                swap_in(next, &slot);

                /* Get data from the header of the block. */
                next  = d_next [slot];
                lines = d_lines [slot];
                data  = data_addr (slot);

                /* Copy each line of the block. */
                count = 0;
                while (lines--) {

                        /* Get length of the line. */
                        length = b_getnum(data + count);

                        /* Skip over length field. */
                        count = count + 2;

                        /* Copy each char of the line. */
                        while (length--) {
                                c = data [count++];
                                write1(c, &out);
                        }

                        /* Add CR and LF at end. */
                        write1(CR, &out);
                        write1(LF, &out);
                }
        }

        /* Force an end of file mark. */
        write1(CPMEOF, &out);

        /* Flush the buffer and close the file. */
        write_flush(out);
        sysclose(b_user_fd);

        /* Kludge:  go to line 1 for a reference point. */
        swap_in(b_head, &b_slot);
        b_line = b_start = 1;



        Write one character to the user's file.
        i is the current position in the file buffer.


/* int */
write1(c, i)
char  ;
int *i;
{
        b_buff [(*i)++] = c;
        if (*i == CPM_SIZE) {

                if (syswrite(b_user_fd, b_buff, 1) !=
                        disk_error("write failed");
                }
                *i = 0;
        }
}

/*
        Flush b_buff to the user's file.
*/

write_flush(i)
int i;
{
        if (i == 0) {
                return;
        }
        if (syswrite(b_user_fd, b_buff, i) !=
                disk_error("flush failed");
        }
}
```

```
/*
        RED buffer routines -- small-C version
        Part 4 -- debugging routines.

        Source:  red13.sc
        Version: August 29, 1982;  February 26, 1983

        Copyright (C) 1983 by Edward K. Ream
*/

/*
        You may omit this file if you need to save space.
        Just delete the calls to check_block() in the
        file red12.sc.  Then delete the #include red13.sc
        statement in file red.sc.
*/

/*
        The observer effect:  None of the routines of this file
        should ever call swap_in() because swap_in() causes
        all kinds changes to the data structures which these
        routines are trying to print out.
*/

/*
        Dump global varialbes, all resident slots and
        the current block.
*/

bufdump()
{
        dump_vars();
        dump_memory();
        dump_block();

}
cant_happen(message)
char *message;
{
        pp(message);
        ppl(":  can't happen");

        bufdump();
        exit();
}
/* Check the current block for consistency. */

check_block(message)
char *message;
{
        int count, i, total;
        char *cp;

        if ( (b_slot == ERROR)  ||
             (b_line < 0)       ||
             (b_line > b_max_line + 1)
           ) {

                ppl("check block 1");
                cant_happen(message);
        }

        if ( (d_lines [b_slot] < 0)            ||
             (d_lines [b_slot] >= BUFF_SIZE)   ||
             (d_avail [b_slot] < 0)            ||
             (d_avail [b_slot] > BUFF_SIZE)
           ) {

                ppl("check block 2");
                cant_happen(message);
        }

        /* Make sure there are at least enough lines. */
        cp     = data_addr (b_slot);
        total = 0;
        i = 0;
        while (i++ < d_lines [b_slot]) {
                count = b_getnum(cp);
                total = total + count + 2;
                cp    = cp + count + 2;

                if ( (count < 0)          |
                     (count > BUFF_SIZE)  |
                     (total > BUFF_SIZE)
                   ) {

                        ppl("check block 3");
                        cant_happen(message);
                }
        }
}
/* Dump the current block. */

dump_block()
{
        int lines, c, count, i, j, total;
        char *cp;

        ppl("");
        ppl("Dump of current block:");
        ppl("");

        lines = d_lines [b_slot];
        cp    = data_addr (b_slot);
        total = 0;

        i = 0;
        while ( (i < lines) & (total <= BUFF_SIZE) ) {
                count = b_getnum(cp);
                total = total + 2;
                cp    = cp + 2;

                pp("line ");
                ppdec(i + 1, 3);
                pp(", count ");
                ppdec(count, 3);
                pp(", total ");
                ppdec(total, 3);
                pp(":  ");

                if ( (count < 0) | (total < 0) ) {
                        return;
                }

                /* Print each line of the block. */

                 j = 0;
                 if (count > 50) {
                        ppl("");
                 }
                 while ( (j < count) & (j < 80) ) {

                        c = cp [j] & 127;

                        if (c == TAB) {
                                ppc(' ');
                        }
                        else if (c < 32) {
                                ppc('^');
                                ppc(c + 64);
                        }
                        else {
                                ppc(c);
                        }

                        total++;
                        if (total >= BUFF_SIZE) {
                                break;
                        }
                        j++;
                }
                ppl("");
                cp = cp + count;

                i++;
        }
}
/* Dump all the resident slots. */

dump_memory()
{
        int             i;

        ppl("Dump of slots:").
        ppl("");

        i = 0;
        while (i < NSLOTS) {

                pp("slot ");
                ppdec(i, 2);
                pp(", diskp ");
                ppdec(d_diskp [i], 6).
                pp(", back ").
                ppdec(d_back [i], 6);
                pp(", next ");
                ppdec(d_next [i], 6);
                pp(", avail ");
                ppdec(d_avail [i], 6).
                pp(", lines ");
                ppdec(d_lines [i], 6);
                pp(", lru ");
                ppdec(d_lru [i], 3).
                pp(", status ");
                ppdec(d_status [i], 3).
                ppl("");
                i++;
        }
}

/* Dump all global variables. */

dump_vars()
{
        pp("slot ");
        ppdec(b_slot, 3);
        pp(", maxdiskp ");
        ppdec(b_max_diskp, 3);
        pp(", line ");
        ppdec(b_line, 5);
        pp(", maxline ");
        ppdec(b_max_line, 5);
        pp(", start ");
        ppdec(b_start, 3);
        pp(", head ");
        ppdec(b_head, 3);
        pp(", tail ");
        ppdec(b_tail, 3);
        pp(", free ");
        ppdec(b_free, 3);
        ppl("");
        pp("back ");
        ppdec(d_back [b_slot], 3);
        pp(", next ");
        ppdec(d_next [b_slot], 3);
        pp(", avail ");
        ppdec(d_avail [b_slot], 3).
        pp(", lines ");
        ppdec(d_lines [b_slot], 3);
        pp(", lru ");
        ppdec(d_lru [b_slot], 3).
        pp(", status ");
        ppdec(d_status [b_slot], 3);
        pp(", diskp ");
        ppdec(d_diskp [b_slot], 3);
        ppl("");
}

/* Output a string to the dump device (console)  */

pp(str)
char * str;
{
        while(*str) {
                syscout(*str++);
        }
}
/* Output a string to the dump device, followed by a CR/LF. */

ppl(str)
char * str;
{
        pp(str);
        syscout(13);
        syscout(10);
}
/* Print a decimal number in a field of width w. */

ppdec(number, w)
int number;
int w;
{
        int count, k, zs;
        char c, buffer[10].

        count = 0;
        zs = 0;
        k = 10000;
        if (number < 0) {
                number = -number;
                buffer [count++] = '-';
        }
        while (k >= 1) {
                c = number/k + '0';
                if ( (c != '0') | (k == 1) | (zs == 1)) {
                        zs = 1;
                        buffer [count++] = c;
                }
                number = number%k;
                k = k/10;
        }
        buffer [count] = 0;
        while (count++ < w) {
                syscout(' ');
        }
        pp(buffer);
}


/* Print one character to the dump device (console). */

ppc(c)
char c;
{
        syscout(c);
}
```

End Listings