CHAPTER 7. THE VIRTUAL MEMORY

The source code discussed in this chapter is in KERNEL86.BLK, screens 50 to 56.

7.1. MASS STORAGE AND VIRTUAL MEMORY

Mass storage is a very important and integral part of a computer although we often think of it as a peripheral or an appendage. The computer uses the RAM memory for most of its normal operations, executing programs stored in RAM and operating on data also stored in RAM. However, programs and data must be saved to more permanent and less expensive media before the power to the computer is turned off, or to transfer programs or data from one computer to another. Without mass storage, a computer is just as useful as a video game, operating entirely from the ROM memory with very limited amount of RAM. Most programming languages, by default or by neglect, do not include facilities to deal with the mass storage as part of the language problem. Thus we have to have a huge beast, an operating system, underneath the language to supply the functions necessary to use the mass storage conveniently and effectively.

Charles Moore perceived the need of the facility to use mass storage efficiently to make Forth not only as a programming language but also as a total environment in which the user can describe the solution of his programming problems. At the time he put together Forth, core memory was much more expensive than now and its use had to be optimized at all costs. His design used the mass storage, whether tape or disk, as a direct extension of the core memory. The user can address the mass storage in the same way he addresses the core memory, without worrying about the detailed processes in storing data to disk or retrieving data from disk. This is the concept of 'virtual memory'.

The way virtual memory operates is as follows: The mass storage, tape or disk, is divided into consecutive blocks as the basic storage units, each block consisting of 1024 bytes. The blocks are numbered from 0 to the capacity of the device, and are addressed by the block number. In the core or the RAM memory in the computer, an area called disk buffer is reserved as temporary storage for blocks of data from disk. One disk buffer is also of 1024 bytes and one or more disk buffers can be reserved. When a block of data is needed, it is read from the disk and stored in one of the disk buffers. Data in this disk buffer can then be used or modified, as needed by the program. When all disk buffers are filled and the system needs to read another block, the system will select the least used disk buffer to receive the new block of data. If the data in the selected disk buffer was modified by the program and was marked as 'updated', the contents of this buffer will be written back to the disk before the new block is read in. This way, the data on disk are assured of their integrity and constantly updated as required, while a few disk buffers can fulfill the need to gain access to the entire disk without large overhead.

If you know how to read and write a sector of the disk, it is not a big job to implement the virtual memory system in Forth. Many Forth systems include such functions. These Forth system have no need for an operating system because the functions of an operating system, handling terminal I/O and managing mass storage, are all provided by the Forth system. In this sense, Forth is its own operating system. F83 on the other hand was designed to run under the CP/M or MS-DOS system and uses the file system in CP/M for mass storage. The advantage is that the resulting Forth system can be easily transported from one computer to another, under the umbrella of CP/M or DOS, and that data can be

dealt with within a more manageable file system. The disadvantage is that one cannot address disk at the sector level and the performance is degraded.

7.2. DISK BUFFERS

A set of pointers and constants is needed to construct the virtual memory system by allocating the disk buffers and defining their characteristics:

```
0  CONSTANT  FIRST          The starting address of the disk buffers. The actual address
                            is patched during Forth initialization.
0  CONSTANT  LIMIT          The address above the top disk buffer.  Also patched at
                            initialization.
4  CONSTANT  #BUFFERS       Four disk buffers are allocated in this example.
1024  CONSTANT  B/BUF       1024 bytes per disk buffer.
128   CONSTANT  B/REC       128 bytes per record in CP/M and DOS.
8  CONSTANT  REC/BLK        8 records per block of 1024 bytes.
42  CONSTANT  B/FCB         42 bytes in a file control block
VARIABLE  DISK-ERROR        Storing error code after a disk operation.
#BUFFERS 1+ 8 * 2+   CONSTANT  >SIZE      The size of a buffer-pointer array. Each disk
                            buffer uses 8 bytes in this array to store buffer information:
                        0-1     Block number        2-3     Pointer to file
                    4-5     Buffer address  6-7     Update flag
```

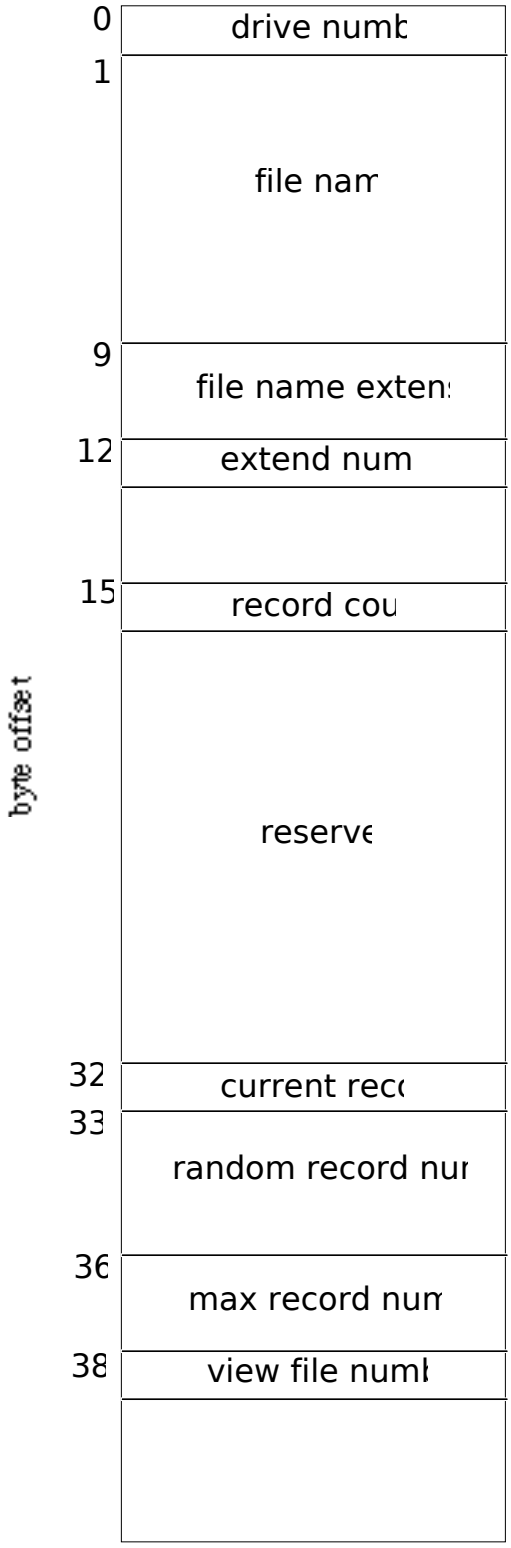
This array is reserved just below the first disk buffer or FIRST. Whenever a block is referenced, its pointer is moved to the head of this array, so that the most recently used buffer is always checked first. This allows the references to multiple disk buffers to be very fast. Disk buffers are 1024 bytes long. No trailing zeros are needed to stop the text interpreter, as in fig-Forth, because the text interpreter in F83 will process only 1024 characters in a buffer. The following words are defined to get pointers to address into this array:

```
: >BUFFERS ( --- addr )     Return the address of the first buffer pointer.
        FIRST               Starting address of first buffer.
        >SIZE               Total bytes in the pointer array.
        -                       First buffer pointer entry.
        ;

: >END              ( --- addr )    Return the address of the last cell in the buffer pointer array.
        FIRST
        2-                      One cell below FIRST.
        ;

: BUFFER#   ( n --- addr )  Return the address of the nth buffer pointer.
        8*                      Offset of the nth buffer pointer.
        >BUFFERS                Origin of the buffer pointer array.
        +       ;
```

Figure 7.1 The File Control Block



7.3. THE FILE CONTROL BLOCK (FCB)

CP/M-DOS programs access the disk files through BDOS calls. However, a program must maintain a special memory array which contains all the information about the file it is using. This memory array is called FCB, File Control Block, usually 36 bytes in length, as shown in Fig. 7.1. The first byte in FCB stores the disk drive code. The next 11 bytes store the file name and extension. Bytes 33, 34, and 35 store the current random record number in read/write operation. F83 reserves 8 more bytes at the end of FCB for some special purposes. A user variable named FILE is used to store the address of the FCB currently used by the Forth system and indicates the current file. All other Forth words doing disk I/O refer to the file pointed to by this user variable FILE. FILE @ leaves the address of the file control block on the stack.

To fully understand the structure of CP/M-DOS files and how they are utilized by programs is beyond the scope of this book. You have to go back to the CP/M manuals where these topics are treated in details. What I shall do here is to go through the F83 disk I/O words and explain there functions. We only have to know a small portion of the CP/M to get a working knowledge of the CP/M-DOS files as required by F83 system.

```
CREATE FCB1 B/FCB ALLOT     Create a default file control block in the Nucleus of F83. 42
                            bytes are reserved.

: CLR-FCB   ( fcb --- )     Initialize the current FCB, given the address of the current
                            File Control Block.
      DUP B/FCB ERASE       Clear the entire array to zero.
      1+                    Address of the first byte of the file name.
      11 BLANK              Initialize the name and extension to blanks, as required by
                            CP/M.
      ;

: SET-DMA          ( addr --- )   Set direct memory transfer address.
      26 BDOS DROP                A standard BDOS call.
      ;

: RECORD#  ( fcb --- addr ) Return the address of the 3 byte pointer to the current
                            random record.
      33 +                  Offset to the random record pointer.
      ;

: MAXREC# ( fcb --- addr )  Return the address of the field storing maximum record
                            number in the current FCB.
      38 +  ;

: VIEW#      ( fcb --- addr )   Return the address where the file number for viewing is
                            stored.
      40 +  ;               The last cell in FCB.

: CAPACITY ( --- n )        Return the number of blocks in the current file.
      FILE @                Fcb of current file.
```

```
MAXREC# @       Get the maximum record number.
1+      0               Make it a double number.
8 UM/MOD                Unsigned mixed division.
NIP                     Discard the remainder.
;
```

```
VARIABLE DISK-ERROR             A variable storing the record number out-of-range flag.

: IN-RANGE ( fcb --- fcb )      Make sure that the current random record is within range.
                                Abort if it is not.
        DUP MAXREC# @           Maximum record in the file.
        OVER RECORD# @          Current record number.
        U<                      Do an unsigned comparison.
        DUP DISK-ERROR ! Store the flag in DISK-ERROR# for diagnostics.
        IF 1                    Error process.
                BUFFER# ONSet buffer flag.
                " Out of Range"
                DISK-ABORTAbort if out of range.
        THEN ;
```

7.4. READ AND WRITE DISK FILES

The following words are the fundamental interface to the disk drive through the CP/M-DOS BDOS. They specify the disk drive, the memory address, the sector to be read or written, and they do the reading or writing.

```
: REC-READ      ( fcb --- )     Read one record from the current file. The record number is
                                stored in the field of random record.
        DUP IN-RANGE            Check the random record number.
        33 BDOS                 Call the read random function.
        ?DISK-ERROR             Store the returned error code in DISK-ERROR.
        ;

: REC-WRITE     ( --- )         Write one random record.
        DUP IN-RANGE            Check the random record number.
        34 BDOS                 Write the record from memory.
        ?DISK-ERROR             Store error code.
        ;
```

One CP/M record is 128 bytes long. One Forth block is 1024 bytes long. To read or write one Forth block, we have to do eight consecutive reads or writes. Another thing we have to take care of is that there are four disk buffers allocated in the F83 system. The buffer to be used for disk I/O has to be specified by a pointer to the appropriate entry in the disk buffer pointer array in front of the buffer area.

```
DEFER  READ-BLOCK               Vectored to FILE-READ.
DEFER  WRITE-BLOCK              Vectored to FILE-WRITE.

: SET-IO   ( buffer-pointer-entry --- buffer rec/blk 0 )        Set up common parameters for file
                                reads or writes.
        DUP 2@                  Get the block number, the first cell in a buffer pointer entry.
        REC/BLK *               The record number.
        OVER RECORD# !          Use it as the random record number. Put it in the FCB.        SWAP
4 + @                   Get the address of the disk buffer in the third cell of the
```

buffer pointer entry.

REC/BLK 0 These two parameters are the index and limit for read/write do-loops in FILE-READ and FILE-WRITE.

;

```
: FILE-READ   ( buffer-pointer-entry --- )    Read 1024 bytes from current file to the disk buffer
                                              specified on stack.
        SET-IO                          Set random record number and leave buffer address and loop
                                        parameters on the stack.
        DO                              Repeat 8 times.
                2DUP SET-DMA            Address for one record.
                DUP REC-READ            Read one record.
                1 SWAP RECORD# +!   Increment the random record number for the next read.
        B/REC +                 Address of next buffer area.
        LOOP  2DROP                     Discard the addr on stack.
        ;

: FILE-WRITE   ( buffer-pointer-entry --- )  Write a block to file.
        SET-IO                          Get buffer address and loop parameters.
        DO                              Repeat 8 times.
                2DUP SET-DMA            Record address.
                DUP REC-WRITE           Write one record.
                1 SWAP RECORD# +!   Next random record.
                B/REC +                 Address of next buffer area.
        LOOP  2DROP   ;

: FILE-IO      ( --- )                  Vector block I/O words to file /O words to use CP/M files.   [']
FILE-READ               Get the address of FILE-READ.
        IS READ-BLOCK                   Vector READ-BLOCK to it.
        [']  FILE-WRITE                 Get address of FILE-WRITE.
        IS WRITE-BLOCK                  Vector WRITE-BLOCK.
        ;
```

7.5. DISK BUFFER MANAGEMENT

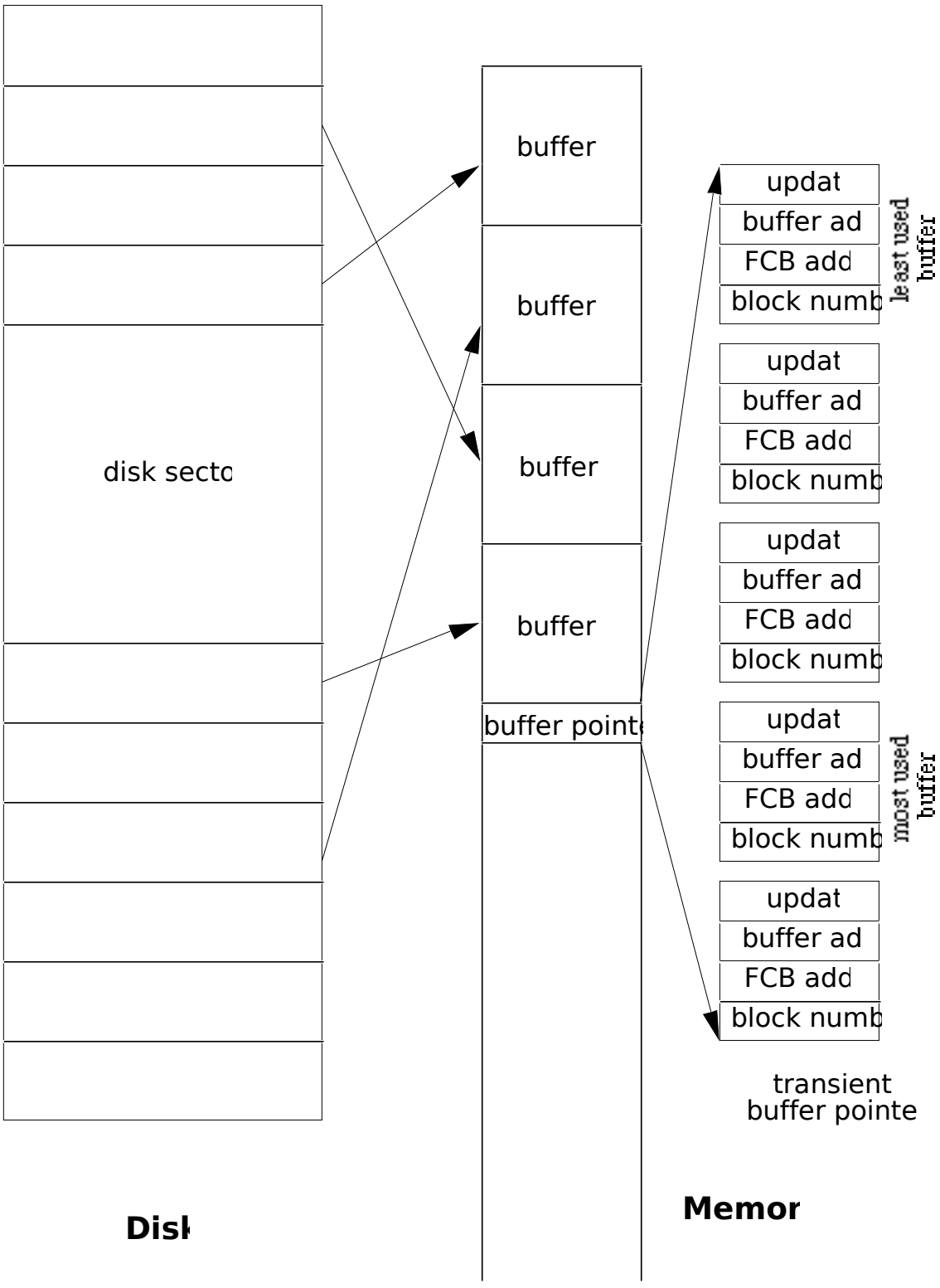
The above set of words allows us to access files on disk. As mentioned earlier, F83 maintains 4 disk buffers in its memory. How are these buffers used? Who decides which buffer is given to which block? When does the block on disk get updated? These are problems we have to face in using a virtual memory system projected into a file system. The following words are designed to deal with these problems. We might call them the 'Virtual Memory Management' in the F83 system.

Let's review what we know about the virtual memory in F83. There are 4 disk buffers, each of 1024 byte length. There is a buffer pointer array with 4 entries, each entry being 8 bytes long. Each entry has four cells containing the block number, the pointer to a file, the buffer address, and the update flag. This array has all the management information for the disk buffers, while the buffers contain the actual data from/ the disk file.

The buffer pointer array is a prioritized structure, in which the first entry points to the most recently used buffer and the last entry points to the least recently used buffer. When a file block is requested, this array is searched. If the disk block is in one of the buffers, its pointer entry is moved to the head of the array. If the disk block is not in the buffers, then the buffer pointed to by the last pointer entry is assigned to the new disk block. However, if the contents of this buffer was modified and the pointer

entry was marked as updated, this buffer will be written back to the disk file before the new disk block is read into this buffer. Thus the disk file is maintained to reflect the current state of any update and modifications, while disk read/write is kept to a minimum. The disk I/O activities are totally transparent to the user, as long as he uses the words BLOCK or BUFFER to access his file.

Figure 7.2 Disk buffer management



```
: LATEST?   ( n fcb --- fcb n | addr f )        Check if block n is the first entry in the buffer pointer
                                                array. If it is, return the buffer address and a false flag, and
                                            exit from the calling word ABSENT?.  If not, return the
                                        block number n with the file control block address.
    DISK-ERROR OFF                      First reset the error flag.
    SWAP OFFSET @ +                     Add the offset block number to obtain the true block number
                                        in the file.
    2DUP                                Leave a copy for return.
    1 BUFFER#                           The first entry in the pointer array.
    2@                                  Get the FCB address and the block number of the buffer
                                        pointed to by the first entry.

    D= IF                               If block n is pointed to by the first entry,
    2DROP                           Drop n and fcb.  They are not needed.
        1 BUFFER#                       Get the address of the entry again.
        4 + @                           Get the address of the disk buffer.
        FALSE                       Push a false flag on top.
        R> DROP                         Discard the top address on the return stack and terminate the
                                        calling word ABSENT?.  The disk buffer was found and
                                    there's no point to search through the pointer array.
    THEN                            Block n is not the first entry.
    ;                                   Return with the block number intact.
```

The most recently referred block is also the block most likely to be referred to the next time. LATEST? thus will cut down much buffer searching overhead and improve significantly the performance of the disk buffer management system.

```
: ABSENT?   ( n fcb --- addr flag )    Search through the buffer pointer array for block n in the
                                            current file.  If it is found, bring the buffer entry to the head
                                            of the array and return the buffer address with a false flag.
                                        If block n is not found in the array, return a dummy address
                                        with a true flag.
    LATEST?                             Is block n same as the first entry in the buffer pointer array?
                                            Exit if so.  Otherwise continue.
    FALSE                               Put a false flag on the data stack as the initial flag before
                                        looping.
    #BUFFERS 1+ 2 DO                    Scan through the buffer pointer array.
        DROP 2DUP                       Get the block number n and fcb address duplicated.
    I BUFFER# 2@                    Get the n and fcb in the pointer array.
        D=  IF                      Is the block number and the fcb a match?
        2DROP I LEAVE               Yes.  Block n is in a buffer. Leave the loop immediately.
    ELSE FALSE THEN             No match.  Put a false flag back.
    LOOP  ?DUP IF                       If a buffer is found to contain the required block, do the
                                            following:
        BUFFER# DUP                     Address of the pointer entry found.
        >BUFFERS                        Starting address of the pointer arrays or the 0th entry.
    8 CMOVE                         Copy this entry to the 0th entry.
        >R                              Save the address of entry found.
```

```
        >BUFFERS        Address of 0th entry.
        DUP 8 +         Address of 1st entry.
        OVER R> SWAP    Current entry address.
        -               Length of entries to be shifted downward by 8 bytes.
CMOVE>          This shift brings the entry with block n to the 1st entry,
                making it the currently used buffer.
```

```
            1 BUFFER#           Address of the 1st entry with block n just found and moved
                                to he 1st entry.
            4 + @               Get its block buffer address.
            FALSE               Put the 'found' flag on data stack.
        ELSE                    No match.  The requested block is not in any of the disk
                                buffers.
            >BUFFERS 2!Store the block number and fcb in the 0th pointer entry.          TRUE
                    Return with a 'not found' flag.
        THEN ;

: >UPDATE  ( --- addr )         Get the address of the update field in the 1st buffer pointer
                                entry.
        1 BUFFER# 6 + ;

: UPDATE    ( --- )             Mark the most recently used buffer as modified.     >UPDATE
                    Address of the update field in the 1st entry.
        ON                      Set it true to indicate that the buffer is modified.
        ;

: DISCARD   ( --- )             Mark the most recently used buffer as unmodified,
                                preventing it from being written back to disk.
        1 >UPDATE !             Store a one in the update field in the 1st entry, marking it as
                            unmodified.
        ;
```

The update cell in the buffer pointer entry is very important in the virtual memory management system. When this cell is set to true (-1), nothing will happen for the moment. However, when this buffer space is assigned to a new disk buffer and before the data in the new block is brought in from the disk, the contents of this buffer will be written back to disk to where it came from. This way, any change we made in the disk buffer will eventually be written back to disk. On the other hand, if the update cell is set to 0 or 1, presumably the contents of the disk buffer is the same as that on the disk, and there is no need of writing the data in the disk buffer back to the disk. Therefore, the new disk block can be brought into this buffer immediately without flushing the old block back to the disk. The disk accessing can thus be reduced while the integrity of data on disk is assured.

```
: MISSING   ( --- )             Discard the least recently used disk buffer.  If this buffer
                                was marked as modified, it is written back to disk.  The first
                                three buffer pointer entries are then shifted down by one
                        entry.  The first entry is made available to the new block to
            be brought in.
        >END 2- @               The address of the update cell in the last buffer pointer entry,
                                which is the least used one.
        0< IF                   If it contains a true flag, write the buffer data back to disk.
        >END 8 -            The block number of the last entry.
            WRITE-BLOCK     Write this buffer back to disk.
            >END 2- OFF Reset the update cell.
        THEN
        >END 4 - @              The buffer address of the last  entry.
```

>BUFFERS 4 + !	Make it the buffer address of the 0th entry.
1 >BUFFERS 6 + !	Mark the 0th entry unmodified.
>BUFFERS	Source address of the down shift of pointer entries.
DUP 8 +	Target address of the down shift.
#BUFFERS 8*	Total bytes to be shifted.

```
        CMOVE>                  Move from the last byte to first.
        ;                       Last pointer entry is discarded. First entry is initialized for
                                a new block of data.

: (BUFFER)  ( n fcb --- addr )  Assign a disk buffer to block n and return the buffer address
                                on the stack.  Block n is not read in from the disk.
        PAUSE                   Allow other tasks a chance of execution.
        ABSENT?                 Is block n already in one of the disk buffers?
        IF                      No.  A buffer has to be allocated to block n.
        MISSING                 Shift the pointer entries.
                1 BUFFER#       Get the first entry.
                4 + @           Fetch the buffer address therein.
        THEN ;

: BUFFER  ( n --- addr )        Do (BUFFER) on the current file.
        FILE @                  Fcb of the current file.
        (BUFFER)                Assign a buffer to the disk block. Write the old block in the
                                buffer to disk if it was updated.
        ;

: (BLOCK)  ( n fcb --- addr ) Return the address of a buffer which contains data from
                        block n.  If block n is not already in one of the buffers, it is
                read in from the disk.
        (BUFFER)                Assign a buffer to the requested block.
        >UPDATE @               Get the update field.
        0>                      Is it 1?
        IF                      Yes.  The block is not in the buffer.
                1 BUFFER# DUP READ-BLOCK    Read it from the disk.
                6 + OFF         Reset the update flag to false.
        THEN ;

VARIABLE FILE                   Pointing to the file control block of the current file.
VARIABLE IN-FILE                Pointing to the file control block of the second opened file or
                                the in-file.

: BLOCK     ( n --- addr )  Read a block from the current file if it is not in the buffer.
                        Return the buffer address.
        FILE @                  Fcb of the current file.
        (BLOCK)                 Read it.
        ;
```

F83 allows you to open and use two files concurrently. The first file is opened with the command OPEN and is referred to as the current file. The second file is open by the command FROM and is called the in-file. The current file is always used as the output file and the in-file is always used as the input file. This way you can copy blocks from one file to another. When OPEN is executed, the invoked file is set to be both the current file and also the in-file so that you can read and write to the same file. The file control block address of the current file is stored in FILE, and that of the in-file is in IN-FILE.

```
: IN-BLOCK ( n --- addr )       Read a block from the in-file, which is the second file
                                opened, to the system.
      IN-FILE @                 Get the fcb of the in-file.
      (BLOCK)                   Read it.
      ;
```

BLOCK is the most important command to communicate with the disk. It is the virtual memory manager. If we need any block of data from disk, just give BLOCK the block number and it will make sure that you have the data in one of the disk buffers. From the address returned by BLOCK, you can access the disk block data using the regular memory accessing commands. If you remember to set the update cell when you change the data in the disk buffer, BLOCK will see to it that the modified data will be written back to disk. This is done using the UPDATE command. You can command BLOCK to ignore any change you made in a disk buffer by the command DISCARD.

In cases that you want to write raw data onto a fresh disk or you do not want to read in the disk block, BUFFER is the word to use because it does not do the disk read. You can use BLOCK for the same purpose, but then you will do a useless disk read operation. When you are writing a large file on to the disk, BUFFER can save you quite some time because only write operations are performed.

7.5. SAVING DISK BUFFERS TO DISK FILES

BLOCK and BUFFER will write to disk only when disk buffers are full and new disk blocks are requested. If the computer is turned off, the buffers will be lost because their contents do not have a chance of being written back. The following commands force the system to write all the updated buffers back to disk. They are highly recommended, especially when you are doing editing work.

```
: SAVE-BUFFERS   ( --- )    Write back all the updated buffers to disk and then mark
                            them all as unmodified.
        1 BUFFER#                   Address of the 1st pointer entry.
        #BUFFERS 0 DO               Scan all the pointer entries.
                DUP @               Get the block number.
                1+ IF               Block number cells were initialized to -1 as empty buffers.
                                    Make sure the buffer is not empty.
                        DUP 6 +     Address of the update cell.
                        @ IF        Is the buffer updated?
                                    DUP WRITE-BLOCK         Yes.  Write it back to disk.
                                    DUP 6 + OFF             Reset the update cell.
                        THEN
                        8 +         Address of the next entry.
                THEN
        LOOP  DROP                  Discard the entry address.
        ;

: EMPTY-BUFFERS   ( --- )           First wipe out the data in the buffers.  Initialize the buffer
                                    pointers to point to the right addresses in memory and reset
                            all the update cells.
        FIRST LIMIT         Boundary of disk buffers.
        OVER - ERASE                Erase the entire disk buffer area.
        >BUFFERS                    Buffer pointer array.
        #BUFFERS 1+ 8 *             Bytes in the pointer array.
        ERASE                       Clear the pointer array.
        FIRST               1st buffer address.
```

```
1 BUFFER#               Address of the 1st pointer entry.
#BUFFERS 0 DO           Go through all pointer entries.
        DUP ON          Initialize the block number cell.
        4 + 2DUP !      Initialize the buffer address cell.
```

```
            SWAP B/BUF +        Address of the next buffer.
            SWAP 4 +            Address of the next pointer entry.
        LOOP  2DROP ;           Clear the stack.

: FLUSH     ( --- )             Save and empty all the buffers.
        SAVE-BUFFERS
        0 BLOCK DROP            Cheat the CP/M system to defeat its extra bufferring in
                                BIOS. By accessing a dummy block, you can be sure that
                                the old one is flushed out of the pipeline and written to disk.
        EMPTY-BUFFERS    ;
```

Whenever you change the disk, be sure to FLUSH out all the buffers to the old disk. During program development, if you have any concern about losing data or crashing the system, FLUSH the buffers first. If you are absolutely sure that the data in the buffers are corrupted, use EMPTY-BUFFERS to clear the buffers. If you want to throw away only one buffer, use DISCARD immediately after you access this buffer by BLOCK, making it the most recently used buffe