-

The Unix I/O System

Dennis M. Ritchie
Bell Telephone Laboratories

This paper gives an overview of the workings of the Unix I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The Unix Time-sharing System." Moreover the present document is intended to be used in conjunction with a copy of the system code, since it is basically an exegesis of that code.

Device Classes

There are two classes of device: *block* and *character.* The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as a word with the minor device number as the low byte and the major device number as the high byte. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u.* This table is indexed by the file descriptor returned by the *open* or *creat,* and is accessed during a *read, write,* or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat.* This table is per-system because the same instance of an open file must be shared among the several processes which can result from *forks* after the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node. Notice that an entry in the *file* table corresponds precisely to an instance of *open* or *creat;* if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base, u.u_count,* and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using an indirect block) to a physical block number; a block-type special file need not be mapped. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

Character device drivers

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function. Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices which require no setup) the *cdevsw* entry can be given as *nulldev;* if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a flag which is non-zero if the file was open for writing in the process which performs the final *close.*

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u_segflg* is supplied which indicates, if *on,* that *u.u_base* refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers, which work one character at a time, the routine

cpass( )

is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u_count* goes to 0 or an error occurs, when it returns −1. *Cpass* takes care of interrogating *u.u_segflg* and updating *u.u_count.*

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine

iomove(buffer, offset, count, flag)

which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer+offset, count* or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write,* except that *u.u_count* is guaranteed to be non-zero. To return characters to the user, the routine

passc(c)

is available; it takes care of housekeeping like *cpass* and returns −1 as the last character specified by *u.u_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write;* the flag should be *B_READ* but the same cautions apply.

The ‘‘special-functions’’ routine is invoked by the *stty* and *gtty* system calls as follows:

*(*p) (dev, v)*

where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u_arg[0...2].*

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the "new PS" word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their "top half" (read/write) and "bottom half" (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc.* A queue header has the structure

```
struct {
        int     c_cc;   /* character count */
        char    *c_cf;  /* first character */
        char    *c_cl;  /* last character */
} queue;
```

A character is placed on the end of a queue by

putc(c, &queue)

where *c* is the character and *queue* is the queue header. The routine returns −1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by

getc(&queue)

which returns either the (non-negative) character or −1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call

sleep(event, priority)

causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority.*

The call

wakeup(event)

indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 127 to −127; a higher numerical value indicates a less-favored scheduling situation. A process sleeping at negative priority cannot be terminated for any reason, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with negative priority on an event which might never occur. On the other hand, calls to *sleep* with non-negative priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area "***u.***" should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup,* (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep(&lbolt, priority)*

may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines

spl4( ), spl5( ), spl6( ), spl7( )

are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then

timeout(func, arg, interval)

will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style

*(*func)(arg)*

Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

An example

The driver for the paper-tape reader/punch is worth examining as a fairly simple example of many of the techniques used in writing character device handlers. The *pc11* structure contains a state (used for the reader), an input queue, and an output queue. A structure, rather than three individual variables, was used to cut down on the number of external symbols which might be confused with symbols in other routines.

When the file is opened for reading, the *open* routine checks to see if its state is not *CLOSED;* if so an error is returned since it is considered a bad idea to let several people read one tape simultaneously. The state is set to *WAITING,* the interrupt is enabled, and a character is requested. The reason for this gambit is that there is no direct way to determine if there is any tape in the reader or if the reader is on-line. In these situations an interrupt will occur immediately and an error indicated. As will be seen, the interrupt routine ignores errors if the state is *WAITING,* but if a good character comes in while in the *WAITING* state the interrupt routine sets the state to *READING.* Thus *open* loops until the state changes, meanwhile sleeping on the "lightning bolt" *lbolt.* If it did not sleep at all, it would prevent any other process from running until the reader came on-line; if it depended on the interrupt routine to wake it up, the effect would be the same, since the error interrupt is almost instantaneous.

The open-write case is much simpler; the punch is enabled and a 100-character leader is punched using *pcleader.*

The *close* routine is also simple; if the reader was open, any uncollected characters are flushed, the interrupt is turned off, and the state is set to *CLOSED.* In the write case a 100-character trailer is punched. The routine has a bug in that if both the reader and punch are open *close* will be called only once, so that either the leftover characters are flushed or the trailer is punched, but not both. It is hard to see how to fix this problem except by making the reader and punch separate devices.

The *pcread* routine tries to pick up characters from the input queue and passes them back until the user's read call is satisfied. If there are no characters it checks whether the state has gone to *EOF,* which means that the interrupt routine detected an error in the *READ* state (assumed to indicate the end of the tape). If so, *pcread* returns; either during this call or the next one no characters will be passed back, indicating an end-of-file. If the state is still *READING* the routine enables another character by fiddling the device's reader control register, provided it is not active, and goes to sleep.

When a reader interrupt occurs and the state is *WAITING,* and the device's error bit is set, the interrupt is ignored; if there is no error the state is set to *READING,* as indicated in the discussion of *pcread.* If the state is *READING* and there is an error, the state is set to *EOF;* it is assumed that the error represents the end of the tape. If there is no error, the character is picked up and stored in the input queue. Then, provided the number of characters already in the queue is less than the high-water mark *PCIHWAT,* the reader is enabled again to read another character. This strategy keeps the tape moving without flooding the input queue with unread characters. Finally, the top half is awakened.

Looking again at *pcread,* notice that the priority level is raised by *spl4( )* to prevent interrupts during the loop. This is done because of the possibility that the input queue is empty, and just after the EOF test is made an error interrupt occurs because the tape runs out. Then *sleep* will be called with no possibility of a *wakeup.* In general the processor priority should be raised when a routine is about to sleep awaiting some condition where the presence of the condition, and the consequent *wakeup,* is indicated by an interrupt. The danger is that the interrupt might occur between the test for the condition and the call to *sleep,* so that the *wakeup* apparently never happens.

At the same time it is a bad idea to raise the processor priority level for an extended period of time, since devices with real-time requirements may be shut out so long as to cause an error. The *pcread* routine is perhaps overzealous in this respect, although since most devices have a priority level higher than 4 this difficulty is not very important.

The *pcwrite* routine simply gets characters from the user and passes them to *pcoutput,* which is separated out so that *pcleader* can call it also. *Pcoutput* checks for errors (like out-of-tape) and if none are present makes sure that the number of characters in the output queue does not exceed *PCOHWAT;* if it does, *sleep* is called. Then the character is placed on the output queue. There is a small bug here in that *pcoutput* does not check that the character was successfully put on the queue (all character-queue space might be empty); perhaps in this case it might be a good idea to sleep on the lightning-bolt until things quiet down. Finally *pcstart* is called, which checks to see if the punch is currently busy, and if not starts the punching of the first character on the output queue.

When punch interrupts occur, *pcpint* is called; it starts the punching of the next character on the output queue, and if the number of characters remaining on the queue is less than the low-water mark *PCOLWAT* it wakes up the write routine, which is presumably waiting for the queue to empty.

The Block-device Interface

Handling of block devices is mediated by a collection of routines which manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes which access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks which are being accessed frequently. The main data base for this mechanism is the table of buffers *buf.* Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw, av_back)* which generally maintain a doubly-linked list of blocks which are "free," that is, eligible to be reallocated for another transaction. Buffers which have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Six routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made "busy," so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

Given a pointer to a buffer, the *brelse* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread.* There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required.

Bawrite places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

Bwrite is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more efficiency is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *brelse, bwrite, bawrite,* or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ

This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swap* which have a separate argument which indicates read or write.

B_DONE

This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if 1 that the returned buffer actually contains the data in the requested block.

B_ERROR

This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b_error* byte of the buffer header may contain an error code if it is non-zero. If *b_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b_error;* the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY

This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread,* which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_WANTED

This flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse)* a *wakeup* is given for the block header whenever *B_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

B_ASYNC

This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.

B_DELWRI

This bit is set by *bdwrite* before releasing the buffer. When *getblk,* while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

B_XMEM

This is actually a mask for the pair of bits which contain the high-order two bits of the physical address of the origin

-

of the buffer; these bits are an extension of the 16 address bits elsewhere in the buffer header.

B_RELOC

This bit is currently unused; it previously indicated that a system-wide relocation constant was to be added to the buffer address. It was needed during a period when addresses of data in the system (including the buffers) were mapped by the relocation hardware to a physical address differing from its virtual address.

Block Device Drivers

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address (including extended-memory bits), the block number, a (negative) word count, and the major and minor device number. The rôle of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B_DONE* (and possibly the *B_ERROR)* bits should be set. Then if the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup.* In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record. [However the mechanism has not been integrated into normal I/O even on magtape and is used only in "raw" I/O as discussed below.]

Notice that although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back),* and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list *(av_forw, av_back)* are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers.

iodone(bp) ,

arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B_ERROR* bit has presumably been set.)

When the device conforms to some rather loose standards adhered to by certain DEC hardware, the routine

devstart(bp, devloc, devblk, hbcom)

is useful. Here *bp* is the address of the buffer header, *devloc* is the address of the slot in the device registers which accepts a perhaps-encoded device block number, *devblk* is the block number, and *hbcom* is a quantity to be stored in the high byte of the device's command register. It is understood, when using this routine, that the device registers are laid out in the order

command register
word count
core address
block (or track or sector)

-

where the address of the last corresponds to *devloc*. Moreover, the device should correspond to the RP, RK, and RF devices with respect to its layout of extended-memory bits and structure of read and write commands.

The routine

geterror(bp)

can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed *(B_DONE* has been set).

An example

The RF disk driver is worth studying as the simplest example of a block I/O device. Its *strategy* routine checks to see if the requested block lies beyond the end of the device; the size of the disk, in this instance, is indicated by the minor device number. If the request is plausible, the buffer is placed at the end of the device queue, and if the disk is not running, *rfstart* is called.

Rfstart merely returns if there is nothing to do, but otherwise sets the device-active flag, loads the address extension register, and calls *devstart* to perform the remaining tasks attendant on beginning a data transfer.

When a completion or error interrupt occurs, *rfintr* is called. If an error is indicated, and if the error count has not exceeded 10, the same transaction is reattempted; otherwise the error bit is set. If there was no error or if 10 failing transfers have been issued the queue is advanced and *rfstart* is called to begin another transaction.

Raw Block-device I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the ''appropriate information'' to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by

physio(strat, bp, dev, rw)

whose arguments are the name of the strategy routine *strat,* the buffer pointer *bp,* the device number *dev,* and a read-write flag *rw* whose value is either *B_READ* or *B_WRITE. Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.

The magtape driver is the only one which as of this writing provides a raw I/O capability; given *physio,* the work involved is trivial, and amounts to passing back to the user information on the length of the record read or written. (There is some funniness because the magtape, uniquely among DEC devices, works in bytes, not words.) Putting in raw I/O for disks should be equally trivial except that the disk address has to be carefully checked to make sure it does not overflow from one logical device to another on which the caller may not have write permission.