Case Study : Minix3 Operating System I/O Management

Sailesh Shiwakoti, 076BCT063, Suraj Niroula, 076BCT089, Surya Narayan Chaudhary, 076BCT091

***Abstract*—The case study is on input/output management in MINIX 3 operating system. Most common block device management and specific device driver for RAM, hard disk, and terminal are studied. Further, one Real Time Clock(RTC) driver is written to display the current date.**

***Index Terms*—MINIX 3, block device, RTC, device driver.**

INTRODUCTION

One of the main functions of an operating system is to control all the computer's I/O (Input/Output) devices. It must issue commands to the devices, catch interrupts, and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. To the extent possible, the interface should be the same for all devices (device independence). The I/O code represents a significant fraction of the total operating system.

I. ARCHITECTURE

MINIX 3 is structured in four layers, with each layer performing a well-defined function. The four layers are illustrated in Fig. 1

The kernel in the bottom layer schedules processes and manages the transitions between the ready, running, and blocked states. The kernel also handles all messages between processes.

The clock task is an I/O device driver in the sense that it interacts with the hardware that generates timing signals, but it is not user accessible like.

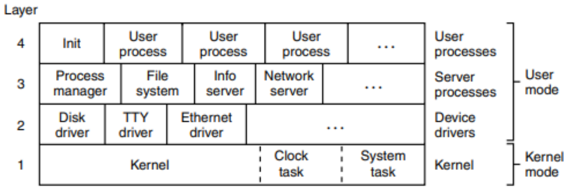


Fig. 1. Four layers of MINIX3

1) System task: One of the main functions of layer 1 is to provide a set of privileged kernel calls to the drivers and servers above it. These include reading and writing I/O ports, copying data between address spaces, and so on. Implementation of these calls is done by the system task. Although the system task and the clock task are compiled into the kernel's address space, they are scheduled as separate processes and have their own call stacks.

2) Servers: The third layer contains servers, processes that provide useful services to the user processes.

The **process manager (PM)** carries out all the MINIX 3 system calls that involve starting or stopping process execution, such as fork, exec, and exit, as well as system calls related to signals, such as alarm and kill, which can alter the execution state of a process. The process manager also is responsible for managing memory, for instance, with the brk system call. The file system (FS) carries out all the file system calls, such as read, mount, and chdir.

The **information server (IS)** handles jobs such as providing debugging and status information about other drivers and servers.

The **reincarnation server (RS)** starts, and if necessary restarts, device drivers that are not loaded into memory at the same time as the kernel. In particular, if a driver fails during operation, the reincarnation server detects this failure, kills the driver if it is not already dead, and starts a fresh copy of the driver, making the system highly fault tolerant.

II. I/O HARDWARE

The programming of many I/O devices is often intimately connected with their internal operation.

A. I/O Devices

I/O devices can be roughly divided into two categories: block devices and character devices. A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 bytes to 32,768 bytes. The other type of I/O device is the character device. A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation.

Device	Date rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K mode	7 KB/sec
Scanner	400 KB/sec
Digital camcorde	4 MB/sec
52x CD-RO	8 MB/sec
FireWire (IEEE 1394)	50 MB/sec
USB 2.0	2.0 60 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
Gigabit Ethernet	125 MB/sec
Serial ATA disk	200 MB/sec
SCSI Ultrawide 4 disk	320 MB/sec

TABLE I
SOME I/O DEVICE AND BUS DATA RATES

B. Device Controllers

I/O units typically consist of a mechanical component and an electronic component. It is often possible to separate the two portions to provide a more modular and general design. The electronic component is called the device controller or adapter. Most personal computers and servers use the bus model of Fig. 2 for communication between the CPU and the controllers.

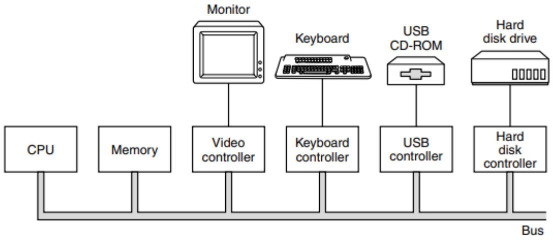


Fig. 2. A model for connecting the CPU, memory, controllers, and I/O devices.

C. Memory Mapped IO

Each controller has a few registers that are used for communicating with the CPU. By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action. By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new command, and so on.

D. Interrupts

Usually, controller registers have one or more status bits that can be tested to determine if an output operation is complete or if new data is available from an input device. A CPU can execute a loop, testing a status bit each time until a device is ready to accept or provide new data. This is called polling or busy waiting.

E. Direct Memory Access

The CPU can request data from an I/O controller one byte at a time but doing so for a device like a disk that produces a large block of data wastes the CPU's time, so a different scheme, called DMA (Direct Memory Access) is often used.

No matter where DMA controller is physically located, the DMA controller has access to the system bus independent of the CPU, as shown in Fig. 3

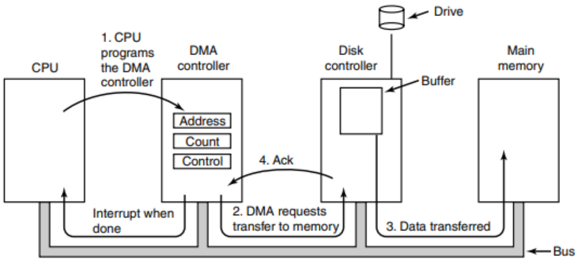


Fig. 3. Operation of a DMA transfer.

III. I/O Software

The goals of I/O software are device independence,uniform naming,error handling, handling synchronous (blocking) and asynchronous (interrupt-driven) transfers, and buffering. Figure 4 shows several layers of I/O system.

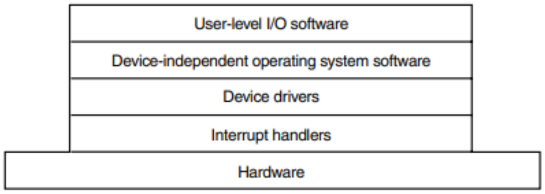


Fig. 4. Layers of the I/O software system.

A. Interrupt Handlers

Many device drivers start some I/O device and then block, waiting for a message to arrive. That message is usually generated by the interrupt handler for the device.

When an interrupt may occur frequently but the amount of I/O handled per interrupt is small, it may pay to make the handler itself do somewhat more work and to postpone sending a message to the driver until a subsequent interrupt, when there is more for the driver to do.

1) Clock Interrupt: The clock's interrupt handler increments a variable, appropriately named real-time, possibly adding a correction for ticks counted during a BIOS call. The handler does some additional very simple arithmetic—it increments counters for user time and billing time, decrements the ticks left counter for the current process, and tests to see if a timer has expired. A message is sent to the clock task only if the current process has used up its quantum or a timer has expired

he clock interrupt handler is unique in MINIX 3, because the clock is the only interrupt driven device that runs in kernel space

There are several different levels of I/O access that might be needed by a user-space device driver.

- A driver might need access to memory outside its normal data space. The memory driver, which manages the RAM disk, is an example of a driver which needs only this kind of access.

- A driver may need to read and write to I/O ports. The machine-level instructions for these operations are available only in kernel mode.
- A driver may need to respond to predictable interrupts. For example, the hard disk driver writes commands to the disk controller, which causes an interrupt to occur when the desired operation is complete.
- A driver may need to respond to unpredictable interrupts. The keyboard driver is in this category. This could be considered a subclass of the preceding item, but unpredictability complicates things

B. Device Drivers

Each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the device driver, is generally written by the device's manufacturer and delivered along with the device

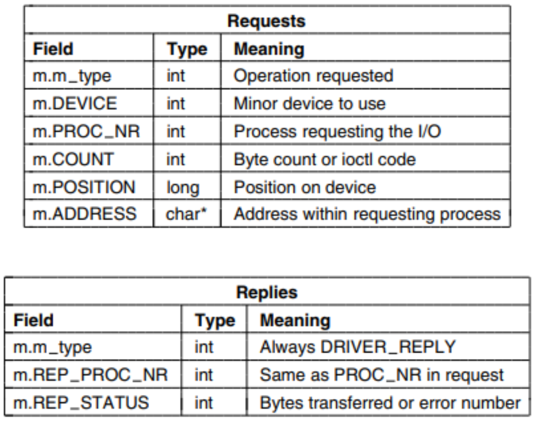
For each class of I/O device present in a MINIX 3 system, a separate I/O device driver is present. O device present in a MINIX 3 system, a separate I/O device driver is present. These drivers are full-fledged processes, each one with its own state, registers, stack, and so on.Device drivers communicate with the file system using the standard message passing mechanism used by all MINIX 3 processes.

The terminal driver source code is organized in a similar way, with the hardware-independent code in tty.c and source code to support different devices, such as memory-mapped consoles, the keyboard, serial lines, and pseudo terminals in separate files.

For groups of devices such as disk devices and terminals, for which there are several source files, there are also header files. *Driver.h* supports all the block device drivers. *Tty.h* provides common definitions for all the terminal devices.

The performance loss due to having most of the operating system run in user space is typically in the range of 5–10% as compared to drivers in UNIX in which they are simply kernel procedures that are called by the kernel-space part of the process. Drivers for RAM disk, hard disk, clock, terminal,RS-232 serial lines, CD-ROMs, various Ethernet adapter, and sound cards are available in MINIX3.

All the block device drivers have been written to get a message, carry it out, and send a reply. The fields for this process is shown in Fig. 5 Among other things, this decision means that these drivers are strictly sequential and do not contain any internal multiprogramming, to keep them simple. When a hardware request has been issued, the driver does a receive operation specifying that it is interested only in accepting interrupt messages, not new requests for work. Any new request messages are just kept waiting until the current work has been done (rendezvous principle). The terminal driver is slightly different, since a single driver services several devices.



Requests		
Field	**Type**	**Meaning**
m.m_type	int	Operation requested
m.DEVICE	int	Minor device to use
m.PROC_NR	int	Process requesting the I/O
m.COUNT	int	Byte count or ioctl code
m.POSITION	long	Position on device
m.ADDRESS	char*	Address within requesting process

Replies		
Field	**Type**	**Meaning**
m.m_type	int	Always DRIVER_REPLY
m.REP_PROC_NR	int	Same as PROC_NR in request
m.REP_STATUS	int	Bytes transferred or error number

Fig. 5. Fields of the messages sent by the file system to the block device drivers and fields of the replies sent back.

C. Device-Independent I/O Software

Although some of the I/O software is device specific, a large fraction of it is device independent. These functions are shown in table below. In MINIX 3 the file system process contains all the device-independent I/O code.

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

TABLE II
FUNCTIONS OF THE DEVICE-INDEPENDENT I/O SOFTWARE.

1) Uniform Interfacing for Device Drivers: A major issue in an operating system is how to make all I/O devices and drivers look more-or-less the same. If disks, printers, monitors, keyboards, etc., are all interfaced in different ways, every time a new peripheral device comes along,the operating system must be modified for the new device. With a standard interface it is much easier to plug in a new driver, providing it conforms to the driver interface. The device-independent software takes care of mapping symbolic device names onto the proper driver. For example, in MINIX 3 a device name, such as /dev/disk0, uniquely specifies the i-node for a special file, and this i-node contains the major device number, which is used to locate the appropriate driver. The i-node also contains the minor device number, which is passed as a parameter to the driver in order to specify the unit to be read or written

2) Buffering: Buffering is also an issue for both block and character devices. For block devices, the hardware generally insists upon reading and writing entire blocks at once, but user processes are free to read and write in arbitrary units. If a user process writes half a block, the operating system will normally keep the data around internally until the rest of the data are written, at which time the block can go out to the disk. For character devices, users can write data to the system faster than it can be output, necessitating buffering. Keyboard input that arrives before it is needed also requires buffering

3) Error Reporting: Errors are far more common in the context of I/O than in any other context. When they occur, the operating system must handle them as best it can. Many errors are device-specific, so only the driver knows what to do (e.g., retry, ignore, or panic). A typical error is caused by a disk block that has been damaged and cannot be read any more. After the driver has tried to read the block a certain number of times, it gives up and informs the device-independent software. How the error is treated from here on is device independent. If the error occurred while reading a user file, it may be sufficient to report the error back to the caller. However, if it occurred while reading a critical system data structure, such as the block containing the bitmap showing which blocks are free, the operating system may have to display an error message and terminate.

4) Allocating and Releasing Dedicated Devices: Some devices, such as CD-ROM recorders, can be used only by a single process at any given moment. It is up to the operating system to examine requests for device usage and accept or reject them, depending on whether the requested device is available or not. A simple way to handle these requests is to require processes to perform opens on the special files for devices directly. If the device is unavailable, the open fails. Closing such a dedicated device then releases it.

5) Device-Independent Block Size: Not all disks have the same sector size. It is up to the device-independent software to hide this fact and provide a uniform block size to higher layers, for example, by treating several sectors as a single logical block. In this way, the higher layers only deal with abstract devices that all use the same logical block size, independent of the physical sector size. Similarly, some character devices deliver their data one byte at a time (e.g., modems), while others deliver theirs in larger units (e.g., network interfaces). These differences may also be hidden.

```
message mess; /* message buffer*/
void io_driver() {
    /* only done once,
    during system init.*/
    initialize();
    while (TRUE) {
     /* wait for a request for work */
        receive(ANY, &mess);
        /* process from whom message came*/
        caller = mess.source;
        switch(mess.type) {
            case READ:
                rcode = dev_read(&mess);
                break;
            case WRITE:
                rcode = dev_write(&mess);
                break;
            /* Other cases go here,
            including OPEN, CLOSE,
            and IOCTL*/
            default: rcode = ERROR;
        }
        mess.type = DRIVER_REPLY;
        /* result code */
        mess.status = rcode;
        /* send reply message
        back to caller */
        send(caller, &mess);
    }
}
```

Program Outline of the main procedure of an I/O device

IV. User-Space I/O Software

1) Library procedures: Although most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs, and even whole programs running outside the kernel. System calls, including the I/O system calls, are normally made by library procedures. When a C program contains the call

```
count = write(fd, buffer, nbytes)
```

the library procedure write will be linked with the program and contained in the binary program present in memory at run time.

2) Spooling: Spooling is a way of dealing with dedicated I/O devices in a multiprogramming system. A special process, called a **daemon**, and a special directory, called a **spooling directory**. To print a file, a process first generates the entire file to be printed and puts it in the spooling directory. It is up to the daemon, which is the only process having permission to use the printer's special file, to print the files in the directory.

The standard MINIX 3 configuration contains one spooler daemon, lpd, which spools and prints files passed to it by the lp command. It also provides a number of daemons that support various network functions.

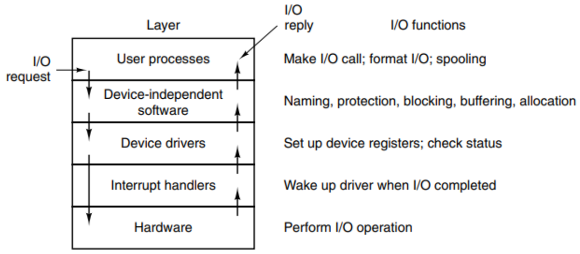

Layer
I/O reply
I/O functions
I/O request
User processes
Device-independent software
Device drivers
Interrupt handlers
Hardware
Make I/O call; format I/O; spooling
Naming, protection, blocking, buffering, allocation
Set up device registers; check status
Wake up driver when I/O completed
Perform I/O operation

Fig. 6. Layers of the I/O system and the main functions of each layer.

V. Deadlock

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

a) Solution: MINIX 3 solves the deadlock by ignoring the problem.

It also avoids the problem in message passing. For instance, user processes are only allowed to use the sendrec messaging method, so a user process should never lock up because it did a receive when there was no process with an interest in sending to it. Servers only use send or sendrec to communicate with device drivers, and device drivers only use send or sendrec to communicate with the system task in the kernel layer. In the rare case where servers must communicate between themselves, such as exchanges between the process manager and the file system as they initialize their parts of the process table, the order of communication is very carefully designed to avoid deadlock. Also, at the very lowest level of the message passing system there is a check to make sure that when a process is about to do a send that the destination process is not trying to the same thing.

VI. BLOCK DEVICES

Each block device driver has to do some initialization. The RAM disk driver has to reserve some memory, the hard disk driver has to determine the parameters of the hard disk hardware, and so on. All of the disk drivers are called individually for hardware-specific initialization. After doing whatever may be necessary, each driver then calls the function containing its main loop. This loop is executed forever; there is no return to the caller. Within the main loop a message is received, a function to perform the operation needed by each message is called, and then a reply message is generated.

There are six possible operations that can be requested of any device driver:

- OPEN
- CLOSE
- READ
- WRITE
- IOCTL
- SCATTERED_IO

OPEN operation should verify that the device is accessible, or return an error message if not, and a CLOSE should guarantee that any buffered data that were written by the caller are completely transferred to their final destination on the device.

Many I/O devices have operational parameters which occasionally must be examined and perhaps changed. IOCTL operations do this.

Except with exceedingly fast disk devices (for example, the RAM disk), satisfactory disk I/O performance is difficult to obtain if all disk requests are for individual blocks, one at a time. A SCATTERED_IO request allows the file system to make a request to read or write multiple blocks. In a SCATTERED_IO request, whether for reading or writing, the list of blocks requested is sorted, and this makes the operation more efficient than handling the requests randomly.

After doing whatever is requested in the message, some sort of cleanup may be necessary, depending upon the nature of the device. For a floppy disk, for instance, this might involve starting a timer to turn off the disk drive motor if another request does not arrive soon.

The first thing each driver does after entering the main loop is to make a call to init_buffer, which assigns a buffer for use in DMA operations.

VII. RAM DRIVER

The MINIX 3 RAM disk driver is actually six closely related drivers in one. Each message to it specifies a minor device as follows: 0: /dev/ram The first special file listed above, /dev/ram, is a true RAM disk. Neither its size nor its origin is built into the driver. They are determined by the file system when MINIX 3 is booted.

/dev/mem /dev/kmem .

The next two minor devices are used to read and write physical memory and kernel memory, respectively. The special file /dev/kmem is like /dev/mem, except that byte 0 of this file is byte 0 of the kernel's data memory.

3: /dev/null The next file in this group, /dev/null, is a special file that accepts data and throws them away. It is commonly used in shell commands when the program being called generates output that is not needed.

4: /dev/boot This is /dev/boot. From the point of view of the device driver it is another block device implemented in RAM, just like /dev/ram. However, it is meant to be initialized by copying a file appended to the boot image after init into memory, rather than starting with an empty block of memory, as is done for /dev/ram.

5: /dev/zero It is sometimes convenient to have a source of zeros. Writing to /dev/zero is like writing to /dev/null; it throws data away. But reading /dev/zero gives you zeros, in any quantity you want, whether a single character or a disk full.

VIII. DISK DRIVER

To deal with all alternatives of disks MINIX 3 allows inclusion of several drivers in the boot image. The MINIX 3 boot monitor allows various boot parameters to be read at startup time. These can be entered by hand, or stored permanently on the disk. a driver that interfaces between MINIX 3 and the ROM BIOS hard disk support. This driver is almost guaranteed to work on any system and can be selected by use of a *label=BIOS* boot parameter.

- A DEV_OPEN request can entail a substantial amount of work, as there are always partitions and may be subpartitions on a hard disk.
- On a CD-ROM a DEV_CLOSE operation also has meaning: it requires that the door be unlocked and the CD-ROM ejected.
- DEV_READ, DEV_WRITE, DEV_GATHER and DEV_SCATTER requests are each handled in two phases, prepare and transfer, as we saw previously. For the hard disk DEV_CANCEL and DEV_SELECT calls are ignored.

A. Disk Arm Scheduling Algorithm

If the disk driver accepts requests one at a time and carries them out in that order, that is, First-Come, First-Served (FCFS), little can be done to optimize seek time. However, most elevators use a different algorithm to reconcile the conflicting goals of efficiency and fairness. They keep moving in the same direction until there are no more outstanding requests in that direction, then they switch directions. This algorithm, known both in the disk world and the elevator world as the elevator algorithm, requires the software to maintain 1 bit: the current direction bit, UP or DOWN. When a request finishes, the disk or elevator driver checks the bit. If it is UP, the arm or cabin is moved to the next highest pending request. If no requests are pending at higher positions, the direction bit is reversed. When the bit is set to DOWN, the move is to the next lowest requested position, if any.

B. *Error Handling*

They are subject to a wide variety of errors. Some of the more common ones are: 1. Programming error (e.g., request for nonexistent sector). 2. Transient checksum error (e.g., caused by dust on the head). 3. Permanent checksum error (e.g., disk block physically damaged). 4. Seek error (e.g., the arm was sent to cylinder 6 but it went to 7). 5. Controller error (e.g., controller refuses to accept commands).

IX. Terminal Driver

A. *Memory-Mapped Terminals*

The first broad category of terminals named in Fig. 3-24 consists of memorymapped terminals. These are an integral part of the computers themselves, especially personal computers. They consist of a display and a keyboard. Memorymapped displays are interfaced via a special memory called a video RAM, which forms part of the computer's address space and is addressed by the CPU the same way as the rest of memory. Also on the video RAM card is a chip called a video controller. This chip pulls bytes out of the video RAM and generates the video signal used to drive the display. Displays are usually one of two types: CRT monitors or flat panel displays. A CRT monitor generates a beam of electrons that scans horizontally across the screen, painting lines on it. Typically the screen has 480 to 1200 lines from top to bottom, with 640 to 1920 points per line. These points are called pixels. The video controller signal modulates the intensity of the electron beam, determining whether a given pixel will be light or dark. Color monitors have three beams, for red, green, and blue, which are modulated independently. A flat panel display works very differently internally, but a CRT-compatible flat-panel display accepts the same synchronization and video signals as a CRT and uses these to control a liquid crystal element at each pixel position.

B. *RS-232 Terminals*

RS-232 terminals are devices containing a keyboard and a display that communicate using a serial interface, one bit at a time (see Fig. 3-27). These terminals use a 9-pin or 25-pin connector, of which one pin is used for transmitting data, one pin is for receiving data, and one pin is ground.

C. *Keyboard*

1) Characters: The first task of the keyboard driver is to collect characters. If every key stroke causes an interrupt, the driver can acquire the character during the interrupt. If interrupts are turned into messages by the low-level software, it is possible to put the newly acquired character in the message. Alternatively, it can be put in a small buffer in memory and the message used to tell the driver that something has arrived. The latter approach is actually safer if a message can be sent only to a waiting process and there is some chance that the keyboard driver might still be busy with the previous character. The first philosophy is character-oriented; the second one is line-oriented. Originally they were referred to as raw mode and cooked mode, respectively. The POSIX standard uses the less-picturesque term canonical mode to describe line-oriented mode. On most systems canonical mode refers to a well-defined configuration. Noncanonical mode is equivalent to raw mode, although many details of terminal behavior can be changed

2) Tab Handling: Another problem is tab handling. All keyboards have a tab key, but displays can handle tab on output. It is up to the driver to compute where the cursor is currently located, taking into account both output from programs and output from echoing, and compute the proper number of spaces to be echoed.

3) Process Pre-emption: It is often necessary to kill a runaway program being debugged. The INTR (CTRL-C) characters can be used for this purpose. In MINIX 3, CTRL-C sends the SIGINT signal to all the processes started up from the terminal. Implementing CTRL-C can be quite tricky. The hard part is getting the information from the driver to the part of the system that handles signals, which, after all, has not asked for this information. CTRL- is similar to CTRL-C, except that it sends the SIGQUIT signal, which forces a core dump if not caught or ignored

4) ioctl: MINIX 3 provides a system call, ioctl, called by ioctl(file_descriptor, request, argp)

that is used to examine and modify the configurations of many I/O devices. This call is used to implement the tcgetattr and tcsetattr functions. The variable request specifies whether the termios structure is to be read or written, and in the latter case, whether the request is to take effect immediately or should be deferred until all currently queued output is complete. The variable argp is a pointer to a termios structure in the calling program.

5) Output: Output is simpler than input, but drivers for RS-232 terminals are radically different from drivers for memory-mapped terminals. The method that is commonly used for RS-232 terminals is to have output buffers associated with each terminal. The buffers can come from the same pool as the input buffers, or be dedicated, as with input. When programs write to the terminal, the output is first copied to the buffers. Similarly, output from echoing is also copied to the buffers. After all the output has been copied to the buffers (or the buffers are full), the first character is output, and the driver goes to sleep.

6) Cursor Positioning: Issue that the driver must deal with on a memory-mapped terminal is cursor positioning. Again, the hardware usually provides some assistance in the form of a register that tells where the cursor is to go.

7) Bell: Finally, there is the problem of the bell. It is sounded by outputting a sine or square wave to the loudspeaker, a part of the computer quite separate from the video RAM.

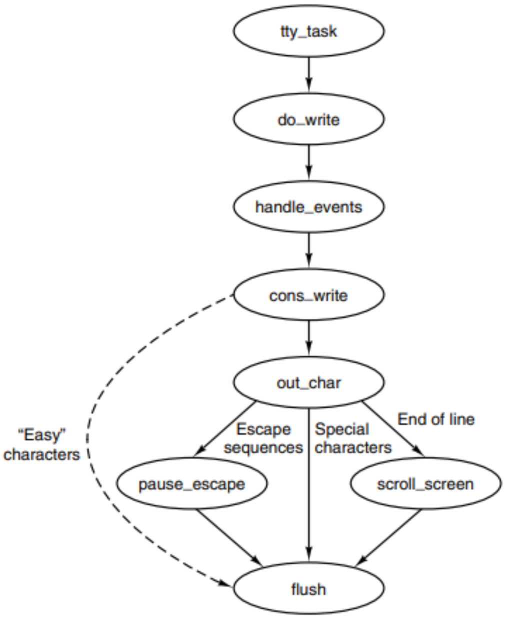


Fig. 7. Major procedures used in terminal output.

D. Implementation

The terminal driver accepts more than a dozen message types. The most important are: 1. Read from the terminal (from FS on behalf of a user process). 2. Write to the terminal (from FS on behalf of a user process). 3. Set terminal parameters for ioctl (from FS on behalf of a user process). 4. A keyboard interrupt has occurred (key pressed or released). 5. Cancel previous request (from FS when a signal occurs). 6. Open a device. 7. Close a device.

Writing a Device Driver

The device driver we wrote reads the seconds, minutes, hours from CMOS hardware and then formats them and prints them. To make a device driver on MINIX 3 we can following steps:

1) Creating header file (our_rtc.h)
2) Creating source file(our_rtc.c) containing open, close and read functions.
3) Making a Makefile
4) Give access to new driver to port of device
5) compile and link the new driver

Conclusion

In this case study we understood the I/O management performed by MINIX 3, generally how I/O devices are made more abstract as we proceed up from real hardware to interrupt handler and all the way up to user programs, particularly how I/O devices RAM, disk and terminal can be implemented and how we can apply modular software development strategy to achieve high reliability, such as self-healing in the time of failure, of the system.