Scan code 29 is one of the modifier codes and must be recognized no matter what other key is pressed, so the CTRL value is returned regardless of any other key that may be pressed. The function keys do not return ordinary ASCII values, and the row for scan code 59 shows symbolically the values (defined in *include/minix/keymap.h*) that are returned for the F1 key in combination with other modifiers. These values are F1: 0x0110, SF1: 0x1010, AF1: 0x0810, ASF1: 0x0C10, and CF1: 0x0210. The last entry shown in the figure, for scan code 127, is typical of many entries near the end of the array. For many keyboards, certainly most of those used in Europe and the Americas, there are not enough keys to generate all the possible codes, and these entries in the table are filled with zeroes.

Loadable Fonts

Early PCs had the patterns for generating characters on a video screen stored only in ROM, but the displays used on modern systems provide RAM on the video display adapters into which custom character generator patterns can be loaded. This is supported by MINIX 3 with a

```
ioctl(0, TIOCSFON, font)
```

ioctl operation. MINIX 3 supports an 80 lines × 25 rows video mode, and font files contain 4096 bytes. Each byte represents a line of 8 pixels that are illuminated if the bit value is 1, and 16 such lines are needed to map each character. However the video display adapter uses 32 bytes to map each character, to provide higher resolution in modes not currently supported by MINIX 3. The *loadfont* command is provided to convert these files into the 8192-byte *font* structure referenced by the ioctl call and to use that call to load the font. As with the keymaps, a font can be loaded at startup time, or at any time during normal operation. However, every video adapter has a standard font built into its ROM that is available by default. There is no need to compile a font into MINIX 3 itself, and the only font support necessary in the kernel is the code to carry out the *TIOCSFON* ioctl operation.

3.8.4 Implementation of the Device-Independent Terminal Driver

In this section we will begin to look at the source code of the terminal driver in detail. We saw when we studied the block devices that multiple drivers supporting several different devices could share a common base of software. The case with the terminal devices is similar, but with the difference that there is one terminal driver that supports several different kinds of terminal device. Here we will start with the device-independent code. In later sections we will look at the device-dependent code for the keyboard and the memory-mapped console display.

Terminal Driver Data Structures

The file *tty.h* contains definitions used by the C files which implement the terminal drivers. Since this driver supports many different devices, the minor device numbers must be used to distinguish which device is being supported on a particular call, and they are defined on lines 13405 to 13409.

Within *tty.h*, the definitions of the *O_NOCTTY* and *O_NONBLOCK* flags (which are optional arguments to the open call) are duplicates of definitions in *include/fcntl.h* but they are repeated here so as not to require including another file. The *devfun_t* and *devfunarg_t* types (lines 13423 and 13424) are used to define pointers to functions, in order to provide for indirect calls using a mechanism similar to what we saw in the code for the main loop of the disk drivers.

Many variables declared in this file are identified by the prefix *tty_*. The most important definition in *tty.h* is the *tty* structure (lines 13426 to 13488). There is one such structure for each terminal device (the console display and keyboard together count as a single terminal). The first variable in the *tty* structure, *tty_events*, is the flag that is set when an interrupt causes a change that requires the terminal driver to attend to the device.

The rest of the *tty* structure is organized to group together variables that deal with input, output, status, and information about incomplete operations. In the input section, *tty_inhead* and *tty_intail* define the queue where received characters are buffered. *Tty_incount* counts the number of characters in this queue, and *tty_eotct* counts lines or characters, as explained below. All device-specific calls are done indirectly, with the exception of the routines that initialize the terminals, which are called to set up the pointers used for the indirect calls. The *tty_devread* and *tty_icancel* fields hold pointers to device-specific code to perform the read and input cancel operations. *Tty_min* is used in comparisons with *tty_eotct*. When the latter becomes equal to the former, a read operation is complete. During canonical input, *tty_min* is set to 1 and *tty_eotct* counts lines entered. During noncanonical input, *tty_eotct* counts characters and *tty_min* is set from the *MIN* field of the *termios* structure. The comparison of the two variables thus tells when a line is ready or when the minimum character count is reached, depending upon the mode. *Tty_*tmr is a timer for this tty, used for the *TIME* field of *termios*.

Since queueing of output is handled by the device-specific code, the output section of *tty* declares no variables and consists entirely of pointers to device-specific functions that write, echo, send a break signal, and cancel output. In the status section the flags *tty_reprint*, *tty_escaped*, and *tty_inhibited* indicate that the last character seen has a special meaning; for instance, when a CTRL-V (LNEXT) character is seen, *tty_escaped* is set to 1 to indicate that any special meaning of the next character is to be ignored.

The next part of the structure holds data about *DEV_READ*, *DEV_WRITE*, and *DEV_IOCTL* operations in progress. There are two processes involved in each of these operations. The server managing the system call (normally FS) is

identified in *tty_incaller* (line 13458). The server calls the tty driver on behalf of another process that needs to do an I/O operation, and this client is identified in *tty_inproc* (line 13459). As described in Fig. 3-33, during a read, characters are transferred directly from the terminal driver to a buffer within the memory space of the original caller. *Tty_inproc* and *tty_in_vir* locate this buffer. The next two variables, *tty_inleft* and *tty_incum*, count the characters still needed and those already transferred. Similar sets of variables are needed for a write system call. For ioctl there may be an immediate transfer of data between the requesting process and the driver, so a virtual address is needed, but there is no need for variables to mark the progress of an operation. An ioctl request may be postponed, for instance, until current output is complete, but when the time is right the request is carried out in a single operation.

Finally, the *tty* structure includes some variables that fall into no other category, including pointers to the functions to handle the *DEV_IOCTL* and *DEV_CLOSE* operations at the device level, a POSIX-style *termios* structure, and a *winsize* structure that provides support for window-oriented screen displays. The last part of the structure provides storage for the input queue itself in the array *tty_inbuf*. Note that this is an array of *u16_t*, not of 8-bit *char* characters. Although applications and devices use 8-bit codes for characters, the C language requires the input function *getchar* to work with a larger data type so it can return a symbolic *EOF* value in addition to all 256 possible byte values.

The *tty_table*, an array of *tty* structures, is declared as *extern* on line 13491. There is one array element for each terminal enabled by the *NR_CONS*, *NR_RS_LINES*, and *NR_PTYS* definitions in *include/minix/config.h*. For the configuration discussed in this book, two consoles are enabled, but MINIX 3 may be recompiled to add additional virtual consoles, one or two 2 serial lines, and up to 64 pseudo terminals.

There is one other *extern* definition in *tty.h*. *Tty_timers* (line 13516) is a pointer used by the timer to hold the head of a linked list of *timer_t* fields. The *tty.h* header file is included in many files and storage for *tty_table* and *tty_timers* is allocated during compilation of *tty.c*.

Two macros, *buflen* and *bufend*, are defined on lines 13520 and 13521. These are used frequently in the terminal driver code, which does much copying of data into and out of buffers.

The Device-Independent Terminal Driver

The main terminal driver and the device-independent supporting functions are all in *tty.c*. Following this there are a number of macro definitions. If a device is not initialized, the pointers to that device's device-specific functions will contain zeroes put there by the C compiler. This makes it possible to define the *tty_active* macro (line 13687) which returns *FALSE* if a null pointer is found. Of course, the initialization code for a device cannot be accessed indirectly if part of its job is to

initialize the pointers that make indirect access possible. On lines 13690 to 13696 are conditional macro definitions to equate initialization calls for RS-232 or pseudo terminal devices to calls to a null function when these devices are not configured. *Do_pty* may be similarly disabled in this section. This makes it possible to omit the code for these devices entirely if it is not needed.

Since there are so many configurable parameters for each terminal, and there may be quite a few terminals on a networked system, a *termios_defaults* structure is declared and initialized with default values (all of which are defined in *include/termios.h*) on lines 13720 to 13727. This structure is copied into the *tty_table* entry for a terminal whenever it is necessary to initialize or reinitialize it. The defaults for the special characters were shown in Fig. 3-29. Figure 3-38 shows the default values for the various flags used. On the following line the *winsize_defaults* structure is similarly declared. It is left to be initialized to all zeroes by the C compiler. This is the proper default action; it means "window size is unknown, use */etc/termcap*."

The final set of definitions before executable code begins are the PUBLIC declarations of global variables previously declared as extern in *tty.h* (lines 13731 to 13735).

Field	Default values
c_iflag	BRKINT ICRNL IXON IXANY
c_oflag	OPOST ONLCR
c_cflag	CREAD CS8 HUPCL
c_lflag	ISIG IEXTEN ICANON ECHO ECHOE

Figure 3-38. Default termios flag values.

The entry point for the terminal driver is *tty_task* (line 13740). Before entering the main loop, a call is made to *tty_init* (line 13752). Information about the host machine that will be needed to initialize the keyboard and the console is obtained by a sys_getmachine kernel call, and then the keyboard hardware is initialized. The routine called for this is *kb_init_once*. It is so named to distinguish it from another initialization routine which is called as part of initialization of each virtual console later on. Finally, a single 0 is printed to exercise the output system and kick anything that does not get initialized until first use. The source code shows a call to *printf*, but this is not the same *printf* used by user programs, it is a special version that calls a local function in the console driver called *putk*.

The main loop on lines 13764 to 13876 is, in principle, like the main loop of any driver—it receives a message, executes a switch on the message type to call the appropriate function, and then generates a return message. However, there are some complications. The first one is that since the last interrupt additional characters may have been read or characters to be written to an output device may be ready. Before attempting to receive a message, the main loop always checks the

tp->tty_events flags for all terminals and *handle_events* is called as necessary to take care of unfinished business. Only when nothing demands immediate attention is a call made to receive.

The diagram showing message types in the comments near the beginning of *tty.c* shows the most often used types. A number of message types requesting specialized services from the terminal driver are not shown. These are not specific to any one device. The *tty_task* main loop checks for these and handles them before checking for device-specific messages. First a check is made for a *SYN_ALARM* message, and, if this is the message type a call is made to *expire_timers* to cause a watchdog routine to execute. Then comes a continue statement. In fact all of the next few cases we will look at are followed by continue. We will say more about this soon.

The next message type tested for is *HARD_INT*. This is most likely the result of a key being pressed or released on the local keyboard. It could also mean bytes have been received by a serial port, if serial ports are enabled—in the configuration we are studying they are not, but we left conditional code in the file here to illustrate how serial port input would be handled. A bit field in the message is used to determine the source of the interrupt.

Next a check is made for *SYS_SIG*. System processes (drivers and servers) are expected to block waiting for messages. Ordinary signals are received only by active processes, so the standard UNIX signaling method does not work with system processes. A *SYS_SIG* message is used to signal a system process. A signal to the terminal driver can mean the kernel is shutting down (*SIGKSTOP*), the terminal driver is being shut down (*SIGTERM*), or the kernel needs to print a message to the console (*SIGKMESS*), and appropriate routines are called for these cases.

The last group of non-device-specific messages are *PANIC_DUMPS*, *DIAGNOSTICS*, and *FKEY_CONTROL*. We will say more about these when we get to the functions that service them.

Now, about the continue statements: in the C language, a continue statement short-circuits a loop, and returns control to the top of the loop. So if any one of the message types mentioned so far is detected, as soon as it is serviced control returns to the top of the main loop, at line 13764, the check for events is repeated, and receive is called again to await a new message. Particularly in the case of input it is important to be ready to respond again as quickly as possible. Also, if any of the message-type tests in the first part of the loop succeeded there is no point in making any of the tests that come after the first switch.

Above we mentioned complications that the terminal driver must deal with. The second complication is that this driver services several devices. If the interrupt is not a hardware interrupt the *TTY_LINE* field in the message is used to determine which device should respond to the message. The minor device number is decoded by a series of comparisons, by means of which *tp* is pointed to the correct entry in the *tty_table* (lines 13834 to 13847). If the device is a pseudo

terminal, *do_pty* (in *pty.c*) is called and the main loop is restarted. In this case *do_pty* generates the reply message. Of course, if pseudo terminals are not enabled, the call to *do_pty* uses the dummy macro defined earlier. One would hope that attempts to access nonexistent devices would not occur, but it is always easier to add another check than to verify that there are no errors elsewhere in the system. In case the device does not exist or is not configured, a reply message with an *ENXIO* error message is generated and, again, control returns to the top of the loop.

The rest of this driver resembles what we have seen in the main loop of other drivers, a switch on the message type (lines 13862 to 13875). The appropriate function for the type of request, *do_read*, *do_write*, and so on, is called. In each case the called function generates the reply message, rather than pass the information needed to construct the message back to the main loop. A reply message is generated at the end of the main loop only if a valid message type was not received, in which case an *EINVAL* error message is sent. Because reply messages are sent from many different places within the terminal driver a common routine, *tty_reply*, is called to handle the details of constructing reply messages.

If the message received by *tty_task* is a valid message type, not the result of an interrupt, and does not come from a pseudo terminal, the switch at the end of the main loop will dispatch to one of the functions *do_read*, *do_write*, *do_ioctl*, *do_open*, *do_close*, *do_select*, or *do_cancel*. The arguments to each of these calls are *tp*, a pointer to a *tty* structure, and the address of the message. Before looking at each of them in detail, we will mention a few general considerations. Since *tty_task* may service multiple terminal devices, these functions must return quickly so the main loop can continue.

However, *do_read*, *do_write*, and *do_ioctl* may not be able to complete all the requested work immediately. In order to allow FS to service other calls, an immediate reply is required. If the request cannot be completed immediately, the *SUSPEND* code is returned in the status field of the reply message. This corresponds to the message marked (3) in Fig. 3-33 and suspends the process that initiated the call, while unblocking the FS. Messages corresponding to (10) and (11) in the figure will be sent later when the operation can be completed. If the request can be fully satisfied, or an error occurs, either the count of bytes transferred or the error code is returned in the status field of the return message to the FS. In this case a message will be sent immediately from the FS back to the process that made the original call, to wake it up.

Reading from a terminal is fundamentally different from reading from a disk device. The disk driver issues a command to the disk hardware and eventually data will be returned, barring a mechanical or electrical failure. The computer can display a prompt upon the screen, but there is no way for it to force a person sitting at the keyboard to start typing. For that matter, there is no guarantee that anybody will be sitting there at all. In order to make the speedy return that is required, *do_read* (line 13953) starts by storing information that will enable the

request to be completed later, when and if input arrives. There are a few error checks to be made first. It is an error if the device is still expecting input to fulfill a previous request, or if the parameters in the message are invalid (lines 13964 to 13972). If these tests are passed, information about the request is copied into the proper fields in the device's *tp->tty_table* entry on lines 13975 to 13979. The last step, setting *tp->tty_inleft* to the number of characters requested, is important. This variable is used to determine when the read request is satisfied. In canonical mode *tp->tty_inleft* is decremented by one for each character returned, until an end of line is received, at which point it is suddenly reduced to zero. In noncanonical mode it is handled differently, but in any case it is reset to zero whenever the call is satisfied, whether by a timeout or by receiving at least the minimum number of bytes requested. When *tp->tty_inleft* reaches zero, a reply message is sent. As we will see, reply messages can be generated in several places. It is sometimes necessary to check whether a reading process still expects a reply; a nonzero value of *tp->tty_inleft* serves as a flag for that purpose.

In canonical mode a terminal device waits for input until either the number of characters asked for in the call has been received, or the end of a line or the end of the file is reached. The *ICANON* bit in the *termios* structure is tested on line 13981 to see if canonical mode is in effect for the terminal. If it is not set, the *termios MIN* and *TIME* values are checked to determine what action to take.

In Fig. 3-31 we saw how *MIN* and *TIME* interact to provide different ways a read call can behave. *TIME* is tested on line 13983. A value of zero corresponds to the left-hand column in Fig. 3-31, and in this case no further tests are needed at this point. If *TIME* is nonzero, then *MIN* is tested. If it is zero, *settimer* is called to start the timer that will terminate the *DEV_READ* request after a delay, even if no bytes have been received. *Tp->tty_min* is set to 1 here, so the call will terminate immediately if one or more bytes are received before the timeout. At this point no check for possible input has yet been made, so more than one character may already be waiting to satisfy the request. In that case, as many characters as are ready, up to the number specified in the read call, will be returned as soon as the input is found. If both *TIME* and *MIN* are nonzero, the timer has a different meaning. The timer is used as an inter-character timer in this case. It is started only after the first character is received and is restarted after each successive character. *Tp->tty_eotct* counts characters in noncanonical mode, and if it is zero at line 13993, no characters have been received yet and the inter-byte timer is inhibited.

In any case, at line 14001, *in_transfer* is called to transfer any bytes already in the input queue directly to the reading process. Next there is a call to *handle_events*, which may put more data into the input queue and which calls *in_transfer* again. This apparent duplication of calls requires some explanation. Although the discussion so far has been in terms of keyboard input, *do_read* is in the device-independent part of the code and also services input from remote terminals connected by serial lines. It is possible that previous input has filled the

RS-232 input buffer to the point where input has been inhibited. The first call to *in_transfer* does not start the flow again, but the call to *handle_events* can have this effect. The fact that it then causes a second call to *in_transfer* is just a bonus. The important thing is to be sure the remote terminal is allowed to send again. Either of these calls may result in satisfaction of the request and sending of the reply message to the FS. *Tp->tty_inleft* is used as a flag to see if the reply has been sent; if it is still nonzero at line 14004, *do_read* generates and sends the reply message itself. This is done on lines 14013 to 14021. (We assume here that no use has been made of the select system call, and therefore there will be no call to *select_retry* on line 14006).

If the original request specified a nonblocking read, the FS is told to pass an *EAGAIN* error code back to original caller. If the call is an ordinary blocking read, the FS receives a *SUSPEND* code, unblocking it but telling it to leave the original caller blocked. In this case the terminal's *tp->tty_inrepcode* field is set to *REVIVE*. When and if the read is later satisfied, this code will be placed in the reply message to the FS to indicate that the original caller was put to sleep and needs to be revived.

Do_write (line 14029) is similar to *do_read*, but simpler, because there are fewer options to be concerned about in handling a write system call. Checks similar to those made by *do_read* are made to see that a previous write is not still in progress and that the message parameters are valid, and then the parameters of the request are copied into the *tty* structure. *Handle_events* is then called, and *tp->tty_outleft* is checked to see if the work was done (lines 14058 to 14060). If so, a reply message already has been sent by *handle_events* and there is nothing left to do. If not, a reply message is generated. with the message parameters depending upon whether or not the original write call was called in nonblocking mode.

The next function, *do_ioctl* (line 14079), is a long one, but not difficult to understand. The body of *do_ioctl* is two switch statements. The first determines the size of the parameter pointed to by the pointer in the request message (lines 14094 to 14125). If the size is not zero, the parameter's validity is tested. The contents cannot be tested here, but what can be tested is whether a structure of the required size beginning at the specified address fits within the segment it is specified to be in. The rest of the function is another switch on the type of ioctl operation requested (lines 14128 to 14225).

Unfortunately, supporting the POSIX-required operations with the ioctl call meant that names for ioctl operations had to be invented that suggest, but do not duplicate, names required by POSIX. Figure 3-39 shows the relationship between the POSIX request names and the names used by the MINIX 3 ioctl call. A *TCGETS* operation services a *tcgetattr* call by the user and simply returns a copy of the terminal device's *tp->tty_termios* structure. The next four request types share code. The *TCSETSW*, *TCSETSF*, and *TCSETS* request types correspond to user calls to the POSIX-defined function *tcsetattr*, and all have the basic action of

POSIX function	POSIX operation	IOCTL type	IOCTL parameter
tcdrain	(none)	TCDRAIN	(none)
tcflow	TCOOFF	TCFLOW	int=TCOOFF
tcflow	TCOON	TCFLOW	int=TCOON
tcflow	TCIOFF	TCFLOW	int=TCIOFF
tcflow	TCION	TCFLOW	int=TCION
tcflush	TCIFLUSH	TCFLSH	int=TCIFLUSH
tcflush	TCOFLUSH	TCFLSH	int=TCOFLUSH
tcflush	TCIOFLUSH	TCFLSH	int=TCIOFLUSH
tcgetattr	(none)	TCGETS	termios
tcsetattr	TCSANOW	TCSETS	termios
tcsetattr	TCSADRAIN	TCSETSW	termios
tcsetattr	TCSAFLUSH	TCSETSF	termios
tcsendbreak	(none)	TCSBRK	int=duration

Figure 3-39. POSIX calls and IOCTL operations.

copying a new *termios* structure into a terminal's *tty* structure. The copying is done immediately for *TCSETS* calls and may be done for *TCSETSW* and *TCSETSF* calls if output is complete, by a *sys_vircopy* kernel call to get the data from the user, followed by a call to *setattr*, on lines 14153 to 14156. If *tcsetattr* was called with a modifier requesting postponement of the action until completion of current output, the parameters for the request are placed in the terminal's *tty* structure for later processing if the test of *tp->tty_outleft* on line 14139 reveals output is not complete. *Tcdrain* suspends a program until output is complete and is translated into an ioctl call of type *TCDRAIN*. If output is already complete, it has nothing more to do. If output is not complete, it also must leave information in the *tty* structure.

The POSIX *tcflush* function discards unread input and/or unsent output data, according to its argument, and the ioctl translation is straightforward, consisting of a call to the *tty_icancel* function that services all terminals, and/or the device-specific function pointed to by *tp->tty_ocancel* (lines 14159 to 14167). *Tcflow* is similarly translated in a straightforward way into an ioctl call. To suspend or restart output, it sets a *TRUE* or *FALSE* value into *tp->tty_inhibited* and then sets the *tp->tty_events* flag. To suspend or restart input, it sends the appropriate *STOP* (normally CTRL-S) or *START* (CTRL-Q) code to the remote terminal, using the device-specific echo routine pointed to by *tp->tty_echo* (lines 14181 to 14186).

Most of the rest of the operations handled by *do_ioctl* are handled in one line of code, by calling an appropriate function. In the cases of the *KIOCSMAP* (load keymap) and *TIOCSFON* (load font) operations, a test is made to be sure the

device really is a console, since these operations do not apply to other terminals. If virtual terminals are in use the same keymap and font apply to all consoles, the hardware does not permit any easy way of doing otherwise. The window size operations copy a *winsize* structure between the user process and the terminal driver. Note, however, the comment under the code for the *TIOCSWINSZ* operation. When a process changes its window size, the kernel is expected to send a *SIGWINCH* signal to the process group under some versions of UNIX. The signal is not required by the POSIX standard and is not implemented in MINIX 3. However, anyone thinking of using these structures should consider adding code here to initiate this signal.

The last two cases in *do_ioctl* support the POSIX required *tcgetpgrp* and *tcsetpgrp* functions. There is no action associated with these cases, and they always return an error. There is nothing wrong with this. These functions support **job control**, the ability to suspend and restart a process from the keyboard. Job control is not required by POSIX and is not supported by MINIX 3. However, POSIX requires these functions, even when job control is not supported, to ensure portability of programs.

Do_open (line 14234) has a simple basic action to perform—it increments the variable *tp->tty_openct* for the device so it can be verified that it is open. However, there are some tests to be done first. POSIX specifies that for ordinary terminals the first process to open a terminal is the **session leader**, and when a session leader dies, access to the terminal is revoked from other processes in its group. Daemons need to be able to write error messages, and if their error output is not redirected to a file, it should go to a display that cannot be closed.

For this purpose a device called */dev/log* exists in MINIX 3. Physically it is the same device as */dev/console*, but it is addressed by a separate minor device number and is treated differently. It is a write-only device, and thus *do_open* returns an *EACCESS* error if an attempt is made to open it for reading (line 14246). The other test done by *do_open* is to test the *O_NOCTTY* flag. If it is not set and the device is not */dev/log*, the terminal becomes the controlling terminal for a process group. This is done by putting the process number of the caller into the *tp->tty_pgrp* field of the *tty_table* entry. Following this, the *tp->tty_openct* variable is incremented and the reply message is sent.

A terminal device may be opened more than once, and the next function, *do_close* (line 14260), has nothing to do except decrement *tp->tty_openct*. The test on line 14266 foils an attempt to close the device if it happens to be */dev/log*. If this operation is the last close, input is canceled by calling *tp->tty_icancel*. Device-specific routines pointed to by *tp->tty_ocancel* and *tp->tty_close* are also called. Then various fields in the *tty* structure for the device are set back to their default values and the reply message is sent.

The last message type handler we will consider is *do_cancel* (line 14281). This is invoked when a signal is received for a process that is blocked trying to read or write. There are three states that must be checked:

1. The process may have been reading when killed.
2. The process may have been writing when killed.
3. The process may have been suspended by *tcdrain* until its output was complete.

A test is made for each case, and the general *tp->tty_icancel*, or the device-specific routine pointed to by *tp->tty_ocancel*, is called as necessary. In the last case the only action required is to reset the flag *tp->tty_ioreq*, to indicate the ioctl operation is now complete. Finally, the *tp->tty_events* flag is set and a reply message is sent.

Terminal Driver Support Code

Now that we have looked at the top-level functions called in the main loop of *tty_task*, it is time to look at the code that supports them. We will start with *handle_events* (line 14358). As mentioned earlier, on each pass through the main loop of the terminal driver, the *tp->tty_events* flag for each terminal device is checked and *handle_events* is called if it shows that attention is required for a particular terminal. *Do_read* and *do_write* also call *handle_events.* This routine must work fast. It resets the *tp->tty_events* flag and then calls device-specific routines to read and write, using the pointers to the functions *tp->tty_devread* and *tp->ttydevwrite* (lines 14382 to 14385).

These functions are called unconditionally, because there is no way to test whether a read or a write caused the raising of the flag—a design choice was made here, that checking two flags for each device would be more expensive than making two calls each time a device was active. Also, most of the time a character received from a terminal must be echoed, so both calls will be necessary. As noted in the discussion of the handling of *tcsetattr* calls by *do_ioctl*, POSIX may postpone control operations on devices until current output is complete, so immediately after calling the device-specific *tty_devwrite* function is a good time take care of ioctl operations. This is done on line 14388, where *dev_ioctl* is called if there is a pending control request.

Since the *tp->tty_events* flag is raised by interrupts, and characters may arrive in a rapid stream from a fast device, there is a chance that by the time the calls to the device-specific read and write routines and *dev_ioctl* are completed, another interrupt will have raised the flag again. A high priority is placed on getting input moved along from the buffer where the interrupt routine places it initially. Thus *handle_events* repeats the calls to the device-specific routines as long as the *tp->tty_events* flag is found raised at the end of the loop (line 14389). When the flow of input stops (it also could be output, but input is more likely to make such repeated demands), *in_transfer* is called to transfer characters from the input queue to the buffer within the process that called for the read operation.

In_transfer itself sends a reply message if the transfer completes the request, either by transferring the maximum number of characters requested or by reaching the end of a line (in canonical mode). If it does so, *tp->tty_left* will be zero upon the return to *handle_events*. Here a further test is made and a reply message is sent if the number of characters transferred has reached the minimum number requested. Testing *tp->tty_inleft* prevents sending a duplicate message.

Next we will look at *in_transfer* (line 14416), which is responsible for moving data from the input queue in the driver's memory space to the buffer of the user process that requested the input. However, a straightforward block copy is not possible here. The input queue is a circular buffer and characters have to be checked to see that the end of the file has not been reached, or, if canonical mode is in effect, that the transfer only continues up through the end of a line. Also, the input queue is a queue of 16-bit quantities, but the recipient's buffer is an array of 8-bit characters. Thus an intermediate local buffer is used. Characters are checked one by one as they are placed in the local buffer, and when it fills up or when the input queue has been emptied, *sys_vircopy* is called to move the contents of the local buffer to the receiving process' buffer (lines 14432 to 14459).

Three variables in the *tty* structure, *tp->tty_inleft*, *tp->tty_eotct*, and *tp->tty_min*, are used to decide whether *in_transfer* has any work to do, and the first two of these control its main loop. As mentioned earlier, *tp->tty_inleft* is set initially to the number of characters requested by a read call. Normally, it is decremented by one whenever a character is transferred but it may be abruptly decreased to zero when a condition signaling the end of input is reached. Whenever it becomes zero, a reply message to the reader is generated, so it also serves as a flag to indicate whether or not a message has been sent. Thus in the test on line 14429, finding that *tp->tty_inleft* is already zero is a sufficient reason to abort execution of *in_transfer* without sending a reply.

In the next part of the test, *tp->tty_eotct* and *tp->tty_min* are compared. In canonical mode both of these variables refer to complete lines of input, and in noncanonical mode they refer to characters. *Tp->tty_eotct* is incremented whenever a "line break" or a byte is placed in the input queue and is decremented by *in_transfer* whenever a line or byte is removed from the queue. In other words, it counts the number of lines or bytes that have been received by the terminal driver but not yet passed on to a reader. *Tp->tty_min* indicates the minimum number of lines (in canonical mode) or characters (in noncanonical mode) that must be transferred to complete a read request. Its value is always 1 in canonical mode and may be any value from 0 up to *MAX_INPUT* (255 in MINIX 3) in noncanonical mode. The second half of the test on line 14429 causes *in_transfer* to return immediately in canonical mode if a full line has not yet been received. The transfer is not done until a line is complete so the queue contents can be modified if, for instance, an ERASE or KILL character is subsequently typed in by the user before the ENTER key is pressed. In noncanonical mode an immediate return occurs if the minimum number of characters is not yet available.

A few lines later, *tp->tty_inleft* and *tp->tty_eotct* are used to control the main loop of *in_transfer*. In canonical mode the transfer continues until there is no longer a complete line left in the queue. In noncanonical mode *tp->tty_eotct* is a count of pending characters. *Tp->tty_min* controls whether the loop is entered but is not used in determining when to stop. Once the loop is entered, either all available characters or the number of characters requested in the original call will be transferred, whichever is smaller.

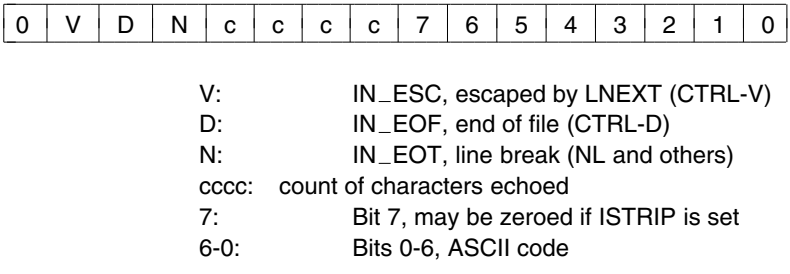

Figure 3-40. The fields in a character code as it is placed into the input queue.

Characters are 16-bit quantities in the input queue. The actual character code to be transferred to the user process is in the low 8 bits. Fig. 3-40 shows how the high bits are used. Three are used to flag whether the character is being escaped (by CTRL-V), whether it signifies end-of-file, or whether it represents one of several codes that signify a line is complete. Four bits are used for a count to show how much screen space is used when the character is echoed. The test on line 14435 checks whether the *IN_EOF* bit (*D* in the figure) is set. This is tested at the top of the inner loop because an end-of-file (CTRL-D) is not itself transferred to a reader, nor is it counted in the character count. As each character is transferred, a mask is applied to zero the upper 8 bits, and only the ASCII value in the low 8 bits is transferred into the local buffer (line 14437).

There is more than one way to signal the end of input, but the device-specific input routine is expected to determine whether a character received is a linefeed, CTRL-D, or other such character and to mark each such character. *In_transfer* only needs to test for this mark, the *IN_EOT* bit (*N* in Fig. 3-40), on line 14454. If this is detected, *tp->tty_eotct* is decremented. In noncanonical mode every character is counted this way as it is put into the input queue, and every character is also marked with the *IN_EOT* bit at that time, so *tp->tty_eotct* counts characters not yet removed from the queue. The only difference in the operation of the main loop of *in_transfer* in the two different modes is found on line 14457. Here *tp->tty_inleft* is zeroed in response to finding a character marked as a line break, but only if canonical mode is in effect. Thus when control returns to the top of the loop, the loop terminates properly after a line break in canonical mode, but in non-canonical line breaks are ignored.

When the loop terminates there is usually a partially full local buffer to be transferred (lines 14461 to 14468). Then a reply message is sent if *tp->tty_inleft* has reached zero. This is always the case in canonical mode, but if noncanonical mode is in effect and the number of characters transferred is less than the full request, the reply is not sent. This may be puzzling if you have a good enough memory for details to remember that where we have seen calls to *in_transfer* (in *do_read* and *handle_events*), the code following the call to *in_transfer* sends a reply message if *in_transfer* returns having transferred more than the amount specified in *tp->tty_min*, which will certainly be the case here. The reason why a reply is not made unconditionally from *in_transfer* will be seen when we discuss the next function, which calls *in_transfer* under a different set of circumstances.

That next function is *in_process* (line 14486). It is called from the device-specific software to handle the common processing that must be done on all input. Its parameters are a pointer to the *tty* structure for the source device, a pointer to the array of 8-bit characters to be processed, and a count. The count is returned to the caller. *In_process* is a long function, but its actions are not complicated. It adds 16-bit characters to the input queue that is later processed by *in_transfer*.

There are several categories of treatment provided by *in_transfer*.

1. Normal characters are added to the input queue, extended to 16 bits.
2. Characters which affect later processing modify flags to signal the effect but are not placed in the queue.
3. Characters which control echoing are acted upon immediately without being placed in the queue.
4. Characters with special significance have codes such as the *EOT* bit added to their high byte as they are placed in the input queue.

Let us look first at a completely normal situation: an ordinary character, such as "x" (ASCII code 0x78), typed in the middle of a short line, with no escape sequence in effect, on a terminal that is set up with the standard MINIX 3 default properties. As received from the input device this character occupies bits 0 through 7 in Fig. 3-40. On line 14504 it would have its most significant bit, bit 7, reset to zero if the *ISTRIP* bit were set, but the default in MINIX 3 is not to strip the bit, allowing full 8-bit codes to be entered. This would not affect our "x" anyway. The MINIX 3 default is to allow extended processing of input, so the test of the *IEXTEN* bit in *tp->tty_termios.c_lflag* (line 14507) passes, but the succeeding tests fail under the conditions we postulate: no character escape is in effect (line 14510), this input is not itself the character escape character (line 14517), and this input is not the *REPRINT* character (line 14524).

Tests on the next several lines find that the input character is not the special *_POSIX_VDISABLE* character, nor is it a *CR* or an *NL*. Finally, a positive result:

canonical mode is in effect, this is the normal default (line 14324). However our "x" is not the *ERASE* character, nor is it any of the *KILL*, *EOF* (CTRL-D), *NL*, or *EOL* characters, so by line 14576 still nothing will have happened to it. Here it is found that the *IXON* bit is set, by default, allowing use of the *STOP* (CTRL-S) and *START* (CTRL-Q) characters, but in the ensuing tests for these no match is found. On line 14597 it is found that the *ISIG* bit, enabling the use of the *INTR* and *QUIT* characters, is set by default, but again no match is found.

In fact, the first interesting thing that might happen to an ordinary character occurs on line 14610, where a test is made to see if the input queue is already full. If this were the case, the character would be discarded at this point, since canonical mode is in effect, and the user would not see it echoed on the screen. (The continue statement discards the character, since it causes the outer loop to restart). However, since we postulate completely normal conditions for this illustration, let us assume the buffer is not full yet. The next test, to see if special noncanonical mode processing is needed (line 14616), fails, causing a jump forward to line 14629. Here *echo* is called to display the character to the user, since the *ECHO* bit in *tp->tty_termios.c_lflag* is set by default.

Finally, on lines 14632 to 14636 the character is disposed of by being put into the input queue. At this time *tp->tty_incount* is incremented, but since this is an ordinary character, not marked by the *EOT* bit, *tp->tty_eotct* is not changed.

The last line in the loop calls *in_transfer* if the character just transferred into the queue fills it. However, under the ordinary conditions we postulate for this example, *in_transfer* would do nothing, even if called, since (assuming the queue has been serviced normally and previous input was accepted when the previous line of input was complete) *tp->tty_eotct* is zero, *tp->tty_min* is one, and the test at the start of *in_transfer* (line 14429) causes an immediate return.

Having passed through *in_process* with an ordinary character under ordinary conditions, let us now go back to the start of *in_process* and look at what happens in less ordinary circumstances. First, we will look at the character escape, which allows a character which ordinarily has a special effect to be passed on to the user process. If a character escape is in effect, the *tp->tty_escaped* flag is set, and when this is detected (on line 14510) the flag is reset immediately and the *IN_ESC* bit, bit V in Fig. 3-40, is added to the current character. This causes special processing when the character is echoed—escaped control characters are displayed as "^" plus the character to make them visible. The *IN_ESC* bit also prevents the character from being recognized by tests for special characters.

The next few lines process the escape character itself, the *LNEXT* character (CTRL-V by default). When the *LNEXT* code is detected the *tp->tty_escaped* flag is set, and *rawecho* is called twice to output a "^" followed by a backspace. This reminds the user at the keyboard that an escape is in effect, and when the following character is echoed, it overwrites the "^". The *LNEXT* character is an example of one that affects later characters (in this case, only the very next character). It is not placed in the queue, and the loop restarts after the two calls to

rawecho. The order of these two tests is important, making it possible to enter the *LNEXT* character itself twice in a row, in order to pass the second copy on to a process as actual data.

The next special character processed by *in_process* is the *REPRINT* character (CTRL-R). When it is found a call to *reprint* ensues (line 14525), causing the current echoed output to be redisplayed. The *REPRINT* itself is then discarded with no effect upon the input queue.

Going into detail on the handling of every special character would be tedious, and the source code of *in_process* is straightforward. We will mention just a few more points. One is that the use of special bits in the high byte of the 16-bit value placed in the input queue makes it easy to identify a class of characters that have similar effects. Thus, *EOT* (CTRL-D), *LF*, and the alternate *EOL* character (undefined by default) are all marked by the *EOT* bit, bit D in Fig. 3-40 (lines 14566 to 14573), making later recognition easy.

Finally, we will justify the peculiar behavior of *in_transfer* noted earlier. A reply is not generated each time it terminates, although in the calls to *in_transfer* we have seen previously, it seemed that a reply would always be generated upon return. Recall that the call to *in_transfer* made by *in_process* when the input queue is full (line 14639) has no effect when canonical mode is in effect. But if noncanonical processing is desired, every character is marked with the *EOT* bit on line 14618, and thus every character is counted by *tp->tty_eotct* on line 14636. In turn, this causes entry into the main loop of *in_transfer* when it is called because of a full input queue in noncanonical mode. On such occasions no message should be sent at the termination of *in_transfer*, because there are likely to be more characters read after returning to *in_process*. Indeed, although in canonical mode input to a single read is limited by the size of the input queue (255 characters in MINIX 3), in noncanonical mode a read call must be able to deliver the POSIX-required constant *_POSIX_SSIZE_MAX* number of characters. Its value in MINIX 3 is 32767.

The next few functions in *tty.c* support character input. *Tty_echo* (line 14647) treats a few characters in a special way, but most just get displayed on the output side of the same device being used for input. Output from a process may be going to a device at the same time input is being echoed, which makes things messy if the user at the keyboard tries to backspace. To deal with this, the *tp->tty_reprint* flag is always set to *TRUE* by the device-specific output routines when normal output is produced, so the function called to handle a backspace can tell that mixed output has been produced. Since *tty_echo* also uses the device-output routines, the current value of *tp->tty_reprint* is preserved while echoing, using the local variable *rp* (lines 14668 to 14701). However, if a new line of input has just begun, *rp* is set to *FALSE* instead of taking on the old value, thus assuring that *tp->tty_reprint* will be reset when *echo* terminates.

You may have noticed that *tty_echo* returns a value, for instance, in the call on line 14629 in *in_process*:

```
ch = tty_echo(tp, ch)
```

The value returned by *echo* contains the number of spaces used on the screen for the echo display, which may be up to eight if the character is a *TAB*. This count is placed in the *cccc* field in Fig. 3-40. Ordinary characters occupy one space on the screen, but if a control character (other than *TAB*, *NL*, or *CR* or a *DEL* (0x7F) is echoed, it is displayed as a "^" plus a printable ASCII character and occupies two positions on the screen. On the other hand an *NL* or *CR* occupies zero spaces. The actual echoing must be done by a device-specific routine, of course, and whenever a character must be passed to the device, an indirect call is made using *tp->tty_echo*, as, for instance, on line 14696, for ordinary characters.

The next function, *rawecho*, is used to bypass the special handling done by *echo*. It checks to see if the *ECHO* flag is set, and if it is, sends the character along to the device-specific *tp->tty_echo* routine without any special processing. A local variable *rp* is used here to prevent *rawecho*'s own call to the output routine from changing the value of *tp->tty_reprint*.

When a backspace is found by *in_process*, the next function, *back_over* (line 14721), is called. It manipulates the input queue to remove the previous head of the queue if backing up is possible—if the queue is empty or if the last character is a line break, then backing up is not possible. Here the *tp->tty_reprint* flag mentioned in the discussions of *echo* and *rawecho* is tested. If it is *TRUE*, then *reprint* is called (line 14732) to put a clean copy of the output line on the screen. Then the *len* field of the last character displayed (the *cccc* field of Fig. 3-40) is consulted to find out how many characters have to be deleted on the display, and for each character a sequence of backspace-space-backspace characters is sent through *rawecho* to remove the unwanted character from the screen and have it replaced by a space.

Reprint is the next function. In addition to being called by *back_over*, it may be invoked by the user pressing the *REPRINT* key (CTRL-R). The loop on lines 14764 to 14769 searches backward through the input queue for the last line break. If it is found in the last position filled, there is nothing to do and *reprint* returns. Otherwise, it echos the CTRL-R, which appears on the display as the two character sequence "^R", and then moves to the next line and redisplays the queue from the last line break to the end.

Now we have arrived at *out_process* (line 14789). Like *in_process*, it is called by device-specific output routines, but it is simpler. It is called by the RS-232 and pseudo terminal device-specific output routines, but not by the console routine. *Out_process* works upon a circular buffer of bytes but does not remove them from the buffer. The only change it makes to the array is to insert a *CR* character ahead of an *NL* character in the buffer if the *OPOST* (enable output processing) and *ONLCR* (map NL to CR-NL) bits in *tp->tty_termios.oflag* are both set. Both bits are set by default in MINIX 3. Its job is to keep the *tp->tty_position* variable in the device's *tty* structure up to date. Tabs and backspaces complicate life.

The next routine is *dev_ioctl* (line 14874). It supports *do_ioctl* in carrying out the *tcdrain* function and the *tcsetattr* function when it is called with either the *TCSADRAIN* or *TCSAFLUSH* options. In these cases, *do_ioctl* cannot complete the action immediately if output is incomplete, so information about the request is stored in the parts of the *tty* structure reserved for delayed ioctl operations. Whenever *handle_events* runs, it first checks the *tp->tty_ioreq* field after calling the device-specific output routine and calls *dev_ioctl* if an operation is pending. *Dev_ioctl* tests *tp->tty_outleft* to see if output is complete, and if so, carries out the same actions that *do_ioctl* would have carried out immediately if there had been no delay. To service *tcdrain*, the only action is to reset the *tp->tty_ioreq* field and send the reply message to the FS, telling it to wake up the process that made the original call. The *TCSAFLUSH* variant of *tcsetattr* calls *tty_icancel* to cancel input. For both variants of *tcsetattr*, the *termios* structure whose address was passed in the original call to ioctl is copied to the device's *tp->tty_termios* structure. *Setattr* is then called, followed, as with *tcdrain*, by sending a reply message to wake up the blocked original caller.

Setattr (line 14899) is the next procedure. As we have seen, it is called by *do_ioctl* or *dev_ioctl* to change the attributes of a terminal device, and by *do_close* to reset the attributes back to the default settings. *Setattr* is always called after copying a new *termios* structure into a device's *tty* structure, because merely copying the parameters is not enough. If the device being controlled is now in noncanonical mode, the first action is to mark all characters currently in the input queue with the *IN_EOT* bit, as would have been done when these characters were originally entered in the queue if noncanonical mode had been in effect then. It is easier just to go ahead and do this (lines 14913 to 14919) than to test whether the characters already have the bit set. There is no way to know which attributes have just been changed and which still retain their old values.

The next action is to check the *MIN* and *TIME* values. In canonical mode *tp->tty_min* is always 1; that is set on line 14926. In noncanonical mode the combination of the two values allows for four different modes of operation, as we saw in Fig. 3-31. On lines 14931 to 14933 *tp->tty_min* is first set up with the value passed in *tp->tty_termios.cc*[*VMIN*], which is then modified if it is zero and *tp->tty_termios.cc*[*VTIME*] is not zero.

Finally, *setattr* makes sure output is not stopped if XON/XOFF control is disabled, sends a *SIGHUP* signal if the output speed is set to zero, and makes an indirect call to the device-specific routine pointed to by *tp->tty_ioctl* to do what can only be done at the device level.

The next function, *tty_reply* (line 14952) has been mentioned many times in the preceding discussion. Its action is entirely straightforward, constructing a message and sending it. If for some reason the reply fails, a panic ensues. The following functions are equally simple. *Sigchar* (line 14973) asks MM to send a signal. If the *NOFLSH* flag is not set, queued input is removed—the count of characters or lines received is zeroed and the pointers to the tail and head of the

queue are equated. This is the default action. When a *SIGHUP* signal is to be caught, *NOFLSH* can be set, to allow input and output to resume after catching the signal. *Tty_icancel* (line 15000) unconditionally discards pending input in the way described for *sigchar*, and in addition calls the device-specific function pointed to by *tp->tty_icancel*, to cancel input that may exist in the device itself or be buffered in the low-level code.

Tty_init (line 15013) is called when *tty_task* first starts. It loops through all possible terminals and sets up defaults. Initially, a pointer to *tty_devnop*, a dummy function that does nothing, is set into the *tp->tty_icancel*, *tp->tty_ocancel*, *tp->tty_ioctl*, and *tp->tty_close* variables. *Tty_init* then calls a device-specific initialization functions for the appropriate category of terminal (console, serial line, or pseudo terminal). These set up the real pointers to indirectly called device-specific functions. Recall that if there are no devices at all configured in a particular device category, a macro that returns immediately is created, so no part of the code for a nonconfigured device need be compiled. The call to *scr_init* initializes the console driver and also calls the initialization routine for the keyboard.

The next three functions support timers. A watchdog timer is initialized with a pointer to a function to run when the timer expires. *Tty_timed_out* is that function for most timers set by the terminal task. It sets the events flag to force processing of input and output. *Expire_timers* handles the terminal driver's timer queue. Recall that this is the function called from the main loop of *tty_task* when a *SYN_ALARM* message is received. A library routine, *tmrs_exptimers*, is used to traverse the linked list of timers, expiring and calling the watchdog functions of any that have timed out. On returning from the library function, if the queue is still active a sys_setalarm kernel call is made to ask for another *SYN_ALARM*. Finally, *settimer* (line 15089), sets timers for determining when to return from a read call in noncanonical mode. It is called with parameters of *tty_ptr*, a pointer to a *tty* structure, and *enable*, an integer which represents *TRUE* or *FALSE*. Library functions *tmrs_settimer* and *tmrs_clrtimer* are used to enable or disable a timer as determined by the *enable* argument. When a timer is enabled, the watchdog function is always *tty_timed_out*, described previously.

A description of *tty_devnop* (line 15125) is necessarily longer than its executable code, since it has none. It is a "no-operation'' function to be indirectly addressed where a device does not require a service. We have seen *tty_devnop* used in *tty_init* as the default value entered into various function pointers before calling the initialization routine for a device.

The final item in *tty.c* needs some explanation. Select is a system call used when multiple I/O devices may require service at unpredictable times by a single process. A classic example is a communications program which needs to pay attention to a local keyboard and a remote system, perhaps connected by a modem. The select call allows opening several device files and monitoring all of them to see when they can be read from or written to without blocking. Without

select it is necessary to use two processes to handle two-way communication, one acting as a master and handling communication in one direction, the other a slave handling communication in the other direction. Select is an example of a feature that is very nice to have, but which substantially complicates the system. One of the design goals of MINIX 3 is to be simple enough to be understood with reasonable effort in a reasonable time, and we have to set some limits. For that reason we will not discuss *do_select* (line 15135) and the support routines *select_try* (line 14313) and *select_retry* (line 14348) here.

3.8.5 Implementation of the Keyboard Driver

Now we turn to the device-dependent code that supports the MINIX 3 console, which consists of an IBM PC keyboard and a memory-mapped display. The physical devices that support these are entirely separate: on a standard desktop system the display uses an adapter card (of which there are at least a half-dozen basic types) plugged into the backplane, while the keyboard is supported by circuitry built into the parentboard which interfaces with an 8-bit single-chip computer inside the keyboard unit. The two subdevices require entirely separate software support, which is found in the files *keyboard.c* and *console.c*.

The operating system sees the keyboard and console as parts of the same device, */dev/console*. If there is enough memory available on the display adapter, **virtual console** support may be compiled, and in addition to */dev/console* there may be additional logical devices, */dev/ttyc1*, */dev/ttyc2*, and so on. Output from only one goes to the display at any given time, and there is only one keyboard to use for input to whichever console is active. Logically the keyboard is subservient to the console, but this is manifested in only two relatively minor ways. First, *tty_table* contains a *tty* structure for the console, and where separate fields are provided for input and output, for instance, the *tty_devread* and *tty_devwrite* fields, pointers to functions in *keyboard.c* and *console.c* are filled in at startup time. However, there is only one *tty_priv* field, and this points to the console's data structures only. Second, before entering its main loop, *tty_task* calls each logical device once to initialize it. The routine called for */dev/console* is in *console.c*, and the initialization code for the keyboard is called from there. The implied hierarchy could just as well have been reversed, however. We have always looked at input before output in dealing with I/O devices and we will continue that pattern, discussing *keyboard.c* in this section and leaving the discussion of *console.c* for the following section.

Keyboard.c begins, like most source files we have seen, with several #include statements. One of these is unusual, however. The file *keymaps/us-std.src* (included on line 15218) is not an ordinary header; it is a C source file that results in compilation of the default keymap within *keyboard.o* as an initialized array. The keymap source file is not included in Appendix B because of its size, but some representative entries are illustrated in Fig. 3-37. Following the #include

statements are macros to define various constants. The first group are used in low-level interaction with the keyboard controller. Many of these are I/O port addresses or bit combinations that have meaning in these interactions. The next group includes symbolic names for special keys. On line 15249 the size of the circular keyboard input buffer is symbolically defined as *KB_IN_BYTES*, with a value of 32, and the buffer itself and variables to manage it are defined next. Since there is only one of these buffers care must be taken to make sure all of its contents are processed before virtual consoles are changed.

The next group of variables are used to hold various states that must be remembered to properly interpret a key press. They are used in different ways. For instance, the value of the *caps_down* flag (line 15266) is toggled between *TRUE* and *FALSE* each time the Caps Lock key is pressed. The *shift* flag (line 15264) is set to *TRUE* when either Shift key is pressed and to *FALSE* when both Shift keys are released. The *esc* variable is set when a scan code escape is received. It is always reset upon receipt of the following character.

Map_key0 (line 15297) is defined as a macro. It returns the ASCII code that corresponds to a scan code, ignoring modifiers. This is equivalent to the first column (unshifted) in the keymap array. Its big brother is *map_key* (line 15303), which performs the complete mapping of a scan code to an ASCII code, including accounting for (multiple) modifier keys that are depressed at the same time as ordinary keys.

The keyboard interrupt service routine is *kbd_interrupt* (line 15335), called whenever a key is pressed or released. It calls *scode* to get the scan code from the keyboard controller chip. The most significant bit of the scan code is set when a key release causes the interrupt, such codes could be ignored unless they were one of the modifier keys. However, in the interest of doing as little as possible in order to service an interrupt as quickly as possible, all raw scan codes are placed in the circular buffer and the *tp->tty_events* flag for the current console is raised (line 15350). For purposes of this discussion we will assume, as we did earlier, that no select calls have been made, and that *kbd_interrupt* returns immediately after this. Figure 3-41 shows scan codes in the buffer for a short line of input that contains two upper case characters, each preceded by the scan code for depression of a shift key and followed by the code for the release of the shift key. Initially codes for both key presses and releases are stored.

When a *HARD_INT* from the keyboard is received by *tty_task*, the complete main loop is not executed. A continue statement at line 13795 causes a new iteration of the main loop to begin immediately, at line 13764. (This is slightly simplified, we left some conditional code in the listing to show that if the serial line driver is enabled its user-space interrupt handler could also be called.) When execution transfers to the top of the loop the *tp->tty_events* flag for the console device is now found to be set, and *kb_read* (line 15360), the device-specific routine, is called using the pointer in the *tp->tty_devread* field of the console's *tty* structure.

42	35	163	170	18	146	38	166	38	166	24	152	57	185
L+	h+	h-	L-	e+	e-	l+	l-	l+	l-	o+	o-	SP+	SP-

54	17	145	182	24	152	19	147	38	166	32	160	28	156
R+	w+	w-	R-	o+	o-	r+	r-	l+	l-	d+	d-	CR+	CR-

Figure 3-41. Scan codes in the input buffer, with corresponding key actions below, for a line of text entered at the keyboard. L and R represent the left and right Shift keys. + and - indicate a key press and a key release. The code for a release is 128 more than the code for a press of the same key.

Kb_read takes scan codes from the keyboard's circular buffer and places ASCII codes in its local buffer, which is large enough to hold the escape sequences that must be generated in response to some scan codes from the numeric keypad. Then it calls *in_process* in the hardware-independent code to put the characters into the input queue. On line 15379 *icount* is decremented. The call to *make_break* returns the ASCII code as an integer. Special keys, such as keypad and function keys, have values greater than 0xFF at this point. Codes in the range from *HOME* to *INSRT* (0x101 to 0x10C, defined in file *include/minix/keymap.h*) result from pressing the numeric keypad, and are converted into 3-character escape sequences shown in Fig. 3-42 using the *numpad_map* array.

The sequences are then passed to *in_process* (lines 15392 to 15397). Higher codes are not passed on to *in_process*. Instead, a check is made for the codes for ALT-LEFT-ARROW, ALT-RIGHT-ARROW, and ALT-F1 through ALT-F12, and if one of these is found, *select_console* is called to switch virtual consoles. CTRL-F1 through CTRL-F12 are similarly given special handling. CTRL-F1 shows the mappings of function keys (more on this later). CTRL-F3 toggles between hardware scrolling and software scrolling of the console screen. CTRL-F7, CTRL-F8, and CTRL-F9 generate signals with the same effects as CTRL-\, CTRL-C, and CTRL-U, respectively, except these cannot be changed by the stty command.

Make_break (line 15431) converts scan codes into ASCII and then updates the variables that keep track of the state of modifier keys. First, however, it checks for the magic CTRL-ALT-DEL combination that PC users all know as the way to force a reboot under MS-DOS. Note the comment that it would be better to do this at a lower level. However, the simplicity of MINIX 3 interrupt handling in kernel space makes detecting CTRL-ALT-DEL impossible there, when an interrupt notification is sent the scan code has not yet been read.

An orderly shutdown is desirable, so rather than try to start the PC BIOS routines, a sys_kill kernel call is made to initiate sending a *SIGKILL* signal TO *init*, the parent process of all other processes (line 15448). *Init* is expected to catch

Key	Scan code	"ASCII"	Escape sequence
Home	71	0x101	ESC [H
Up Arrow	72	0x103	ESC [A
Pg Up	73	0x107	ESC [V
–	74	0x10A	ESC [S
Left Arrow	75	0x105	ESC [D
5	76	0x109	ESC [G
Right Arrow	77	0x106	ESC [C
+	78	0x10B	ESC [T
End	79	0x102	ESC [Y
Down Arrow	80	0x104	ESC [B
Pg Dn	81	0x108	ESC [U
Ins	82	0x10C	ESC [@

Figure 3-42. Escape codes generated by the numeric keypad. When scan codes for ordinary keys are translated into ASCII codes the special keys are assigned "pseudo ASCII" codes with values greater than 0xFF.

this signal and interpret it as a command to begin an orderly process of shutting down, prior to causing a return to the boot monitor, from which a full restart of the system or a reboot of MINIX 3 can be commanded.

Of course, it is not realistic to expect this to work every time. Most users understand the dangers of an abrupt shutdown and do not press CTRL-ALT-DEL until something is really going wrong and normal control of the system has become impossible. At this point it is likely that the system may be so disrupted that signaling another process may be impossible. This is why there is a *static* variable *CAD_count* in *make_break*. Most system crashes leave the interrupt system still functioning, so keyboard input can still be received and the terminal driver will remain active. Here MINIX 3 takes advantage of the expected behavior of computer users, who are likely to bang on the keys repeatedly when something does not seem to work correctly (possibly evidence our ancestors really were apes). If the attempt to kill *init* fails and the user presses CTRL-ALT-DEL twice more, a sys_abort kernel call is made, causing a return to the monitor without going through the call to *init*.

The main part of *make_break* is not hard to follow. The variable *make* records whether the scan code was generated by a key press or a key release, and then the call to *map_key* returns the ASCII code to *ch*. Next is a switch on *ch* (lines 15460 to 15499). Let us consider two cases, an ordinary key and a special key. For an ordinary key, none of the cases match, and in the default case (line 15498), the key code is returned if *make* is true. If somehow an ordinary key code is accepted at key release, a value of −1 is substituted here, and this is ignored by

the caller, *kb_read*. A special key, for example *CTRL*, is identified at the appropriate place in the switch, in this case on line 15461. The corresponding variable, in this case *ctrl*, records the state of *make*, and −1 is substituted for the character code to be returned (and ignored). The handling of the *ALT*, *CALOCK*, *NLOCK*, and *SLOCK* keys is more complicated, but for all of these special keys the effect is similar: a variable records either the current state (for keys that are only effective while pressed) or toggles the previous state (for the lock keys).

There is one more case to consider, that of the *EXTKEY* code and the *esc* variable. This is not to be confused with the ESC key on the keyboard, which returns the ASCII code 0x1B. There is no way to generate the *EXTKEY* code alone by pressing any key or combination of keys; it is the PC keyboard's **extended key prefix**, the first byte of a 2-byte scan code that signifies that a key that was not part of the original PC's complement of keys but that has the same scan code, has been pressed. In many cases software treats the two keys identically. For instance, this is almost always the case for the normal "/" key and the gray "/" key on the numeric keyboard. In other cases, one would like to distinguish between such keys. For instance, many keyboard layouts for languages other than English treat the left and right ALT keys differently, to support keys that must generate three different character codes. Both ALT keys generate the same scan code (56), but the *EXTKEY* code precedes this when the right-hand ALT is pressed. When the *EXTKEY* code is returned, the *esc* flag is set. In this case, *make_break* returns from within the switch, thus bypassing the last step before a normal return, which sets *esc* to zero in every other case (line 15458). This has the effect of making the *esc* effective only for the very next code received. If you are familiar with the intricacies of the PC keyboard as it is ordinarily used, this will be both familiar and yet a little strange, because the PC BIOS does not allow one to read the scan code for an ALT key and returns a different value for the extended code than does MINIX 3.

Set_leds (line 15508) turns on and off the lights that indicate whether the Num Lock, Caps Lock, or Scroll Lock keys on a PC keyboard have been pressed. A control byte, *LED_CODE*, is written to an output port to instruct the keyboard that the next byte written to that port is for control of the lights, and the status of the three lights is encoded in 3 bits of that next byte. These operations are, of course, carried out by kernel calls which ask the system task write to the outport ports. The next two functions support this operation. *Kb_wait* (line 15530) is called to determine that the keyboard is ready to receive a command sequence, and *kb_ack* (line 15552) is called to verify that the command has been acknowledged. Both of these commands use busy waiting, continually reading until a desired code is seen. This is not a recommended technique for handling most I/O operations, but turning lights on and off on the keyboard is not going to be done very often and doing it inefficiently does not waste much time. Note also that both *kb_wait* and *kb_ack* could fail, and one can determine from the return code if this happens. Timeouts are handled by limiting the number of retries by means

of a counter in the loop. But setting the light on the keyboard is not important enough to merit checking the value returned by either call, and *set_leds* just proceeds blindly.

Since the keyboard is part of the console, its initialization routine, *kb_init* (line 15572), is called from *scr_init* in *console.c*, not directly from *tty_init* in *tty.c*. If virtual consoles are enabled, (i.e., *NR_CONS* in *include/minix/config.h* is greater than 1), *kb_init* is called once for each logical console. The next function, *kb_init_once* (line 15583), is called just once, as its name implies. It sets the lights on the keyboard, and scans the keyboard to be sure no leftover keystroke is read. Then it initializes two arrays, *fkey_obs* and *sfkey_obs* which are used to bind function keys to the processes that must respond to them. When all is ready, it makes two kernel calls, sys_irqsetpolicy and sys_irqenable to set up the IRQ for the keyboard and configure it to automatically reenable, so a notification message will be sent to *tty_task* whenever a key is pressed or released.

Although we will soon have more opportunities to discuss how function keys work, this is a good place to describe the *fkey_obs* and *sfkey_obs* arrays. Each has twelve elements, since modern PC keyboards have twelve F-keys. The first array is for unmodified F-keys, the second is used when a shifted F-key is detected. They are composed of elements of type *obs_t*, which is a structure that can hold a process number and an integer. This structure and these arrays are declared in *keyboard.c* on lines 15279 to 15281. Initialization stores a special value, symbolically represented as *NONE*, in the *proc_nr* component of the structure to indicate it is not in use. *NONE* is a value outside the range of valid process numbers. Note that the process number is not a *pid*, it identifies a slot in the process table. This terminology may be confusing. But to send a notification a process number rather than a *pid* is used, because process numbers are used to index the *priv* table which determines whether a process is allowed to receive notifications. The integer *events* is also initially set to zero. It will be used to count events.

The next three functions are all rather simple. *Kbd_loadmap* (line 15610) is almost trivial. It is called by *do_ioctl* in *tty.c* to do the copying of a keymap from user space to overwrite the default keymap. The default is compiled by the inclusion of a keymap source file at the start of *keyboard.c*.

From its first release, MINIX has always provided for dumps of various kinds of system information or other special actions in response to pressing the function keys F1, F2, etc., on the system console. This is not a service generally provided in other operating systems, but MINIX was always intended to be a teaching tool. Users are encouraged to tinker with it, which means users may need extra help for debugging. In many cases the output produced by pressing one of the F-keys will be available even when the system has crashed. Figure 3-43 summarizes these keys and their effects.

These keys fall into two categories. As noted earlier, the CTRL-F1 through CTRL-F12 key combinations are detected by *kb_read*. These trigger events that

Key	Purpose
F1	Kernel process table
F2	Process memory maps
F3	Boot image
F4	Process privileges
F5	Boot monitor parameters
F6	IRQ hooks and policies
F7	Kernel messages
F10	Kernel parameters
F11	Timing details (if enabled)
F12	Scheduling queues
SF1	Process manager process table
SF2	Signals
SF3	File system process table
SF4	Device/driver mapping
SF5	Print key mappings
SF9	Ethernet statistics (RTL8139 only)
CF1	Show key mappings
CF3	Toggle software/hardware console scrolling
CF7	Send SIGQUIT, same effect as CTRL-\
CF8	Send SIGINT, same effect as CTRL-C
CF9	Send SIGKILL, same effect as CTRL-U

Figure 3-43. The function keys detected by *func_key()*.

can be handled by the terminal driver. These events are not necessarily display dumps. In fact, currently only CTRL-F1 provides an information display; it lists function key bindings. CTRL-F3 toggles hardware and software scrolling of the console screen, and the others cause signals.

Function keys pressed by themselves or together with the shift key are used to trigger events that cannot be handled by the terminal driver. They may result in notification messages to a server or driver. Because servers and drivers can be loaded, enabled, and disabled after MINIX 3 is already running, static binding of these keys at compilation time is not satisfactory. To enable run-time changes *tty_task* accepts messages of type *FKEY_CONTROL*. *Do_fkey_ctl* (line 15624) services such requests. Request types are *FKEY_MAP*, *FKEY_UNMAP*, or *FKEY_EVENTS*. The first two register or unregister a process with a key specified in a bitmap in the message, and the third message type returns a bitmap of keys belonging to the caller which have been pressed and resets the *events* field for these keys. A server process, the **information server**, (or **IS**) initializes the

settings for processes in the boot image and also mediates generating responses. But individual drivers can also register to respond to a function key. Ethernet drivers typically do this, as a dump that shows packet statistics can be helpful in solving network problems.

Func_key (line 15715) is called from *kb_read* to see if a special key meant for local processing has been pressed. This is done for every scan code received, prior to any other processing. If it is not a function key at most three comparisons are made before control is returned to *kb_read*. If a function key is registered a notification message is sent to the appropriate process. If the process is one that has registered only one key the notification by itself is adequate for the process to know what to do. If a process is the information server or another that has registered several keys, a dialogue is required—the process must send an *FKEY_EVENTS* request to the terminal driver, to be processed by *do_fkey_ctl* which will inform the caller which keys have been active. The caller can then dispatch to the routine for each key that has been pressed.

Scan_keyboard (line 15800) works at the hardware interface level, by reading and writing bytes from I/O ports. The keyboard controller is informed that a character has been read by the sequence on lines 15809 and 15810, which reads a byte, writes it again with the most significant bit set to 1, and then rewrites it with the same bit rest to 0. This prevents the same data from being read on a subsequent read. There is no status checking in reading the keyboard, but there should be no problems in any case, since *scan_keyboard* is only called in response to an interrupt.

The last function in *keyboard.c* is *do_panic_dumps* (line 15819). If invoked as a result of a system panic, it provides an opportunity for the user to use the function keys to display debugging information. The loop on lines 15830 to 15854 is another example of busy waiting. The keyboard is read repeatedly until an ESC is typed. Certainly no one can claim that a more efficient technique is needed after a crash, while awaiting a command to reboot. Within the loop, the rarely-used nonblocking receive operation, nb_receive, is used to permit alternately accepting messages, if available, and testing the keyboard for input, which can be expected to be one of the options suggested in the message

```
Hit ESC to reboot, DEL to shutdown, F-keys for debug dumps
```

printed on entering this function. In the next section we will see the code that implements *do_newkmess* and *do_diagnostics*.

3.8.6 Implementation of the Display Driver

The IBM PC display may be configured as several virtual terminals, if sufficient memory is available. We will examine the console's device-dependent code in this section. We will also look at the debug dump routines that use low-level

services of the keyboard and display. These provide support for limited interaction with the user at the console, even when other parts of the MINIX 3 system are not functioning and can provide useful information even following a near-total system crash.

Hardware-specific support for console output to the PC memory-mapped screen is in *console.c*. The *console* structure is defined on lines 15981 to 15998. In a sense this structure is an extension of the *tty* structure defined in *tty.c*. At initialization the *tp->tty_priv* field of a console's *tty* structure is assigned a pointer to its own *console* structure. The first item in the *console* structure is a pointer back to the corresponding *tty* structure. The components of a *console* structure are what one would expect for a video display: variables to record the row and column of the cursor location, the memory addresses of the start and limit of memory used for the display, the memory address pointed to by the controller chip's base pointer, and the current address of the cursor. Other variables are used for managing escape sequences. Since characters are initially received as 8-bit bytes and must be combined with attribute bytes and transferred as 16-bit words to video memory, a block to be transferred is built up in *c_ramqueue*, an array big enough to hold an entire 80-column row of 16-bit character-attribute pairs. Each virtual console needs one *console* structure, and the storage is allocated in the array *cons_table* (line 16001). As we have done with the *tty* and other structures, we will usually refer to the elements of a *console* structure using a pointer, for example, *cons->c_tty*.

The function whose address is stored in each console's *tp->tty_devwrite* entry is *cons_write* (line 16036). It is called from only one place, *handle_events* in *tty.c*. Most of the other functions in *console.c* exist to support this function. When it is called for the first time after a client process makes a write call, the data to be output are in the client's buffer, which can be found using the *tp->tty_outproc* and *tp->out_vir* fields in the *tty* structure. The *tp->tty_outleft* field tells how many characters are to be transferred, and the *tp->tty_outcum* field is initially zero, indicating none have yet been transferred. This is the usual situation upon entry to *cons_write*, because normally, once called, it transfers all the data requested in the original call. However, if the user wants to slow the process in order to review the data on the screen, he may enter a *STOP* (CTRL-S) character at the keyboard, resulting in raising of the *tp->tty_inhibited* flag. *Cons_write* returns immediately when this flag is raised, even though the write has not been completed. In such a case *handle_events* will continue to call *cons_write*, and when *tp->tty_inhibited* is finally reset, by the user entering a *START* (CTRL-Q) character, *cons_write* continues with the interrupted transfer.

Cons_write's first argument is a pointer to the particular console's *tty* structure, so the first thing that must be done is to initialize *cons*, the pointer to this console's *console* structure (line 16049). Then, because *handle_events* calls *cons_write* whenever it runs, the first action is a test to see if there really is work to be done. A quick return is made if not (line 16056). Following this the main

loop on lines 16061 to 16089 is entered. This loop is similar in structure to the main loop of *in_transfer* in *tty.c*. A local buffer that can hold 64 characters is filled by using the *sys_vircopy* kernel call to get the data from the client's buffer. Following this, the pointer to the source and the counts are updated, and then each character in the local buffer is transferred to the *cons->c_ramqueue* array, along with an attribute byte, for later transfer to the screen by *flush*.

The transfer of characters from *cons->c_ramqueue* can be done in more than one way, as we saw in Fig. 3-35. *Out_char* can be called to do this for each character, but it is predictable that none of the special services of *out_char* will be needed if the character is a visible character, an escape sequence is not in progress, the screen width has not been exceeded, and *cons->c_ramqueue* is not full. If the full service of *out_char* is not needed, the character is placed directly into *cons->c_ramqueue*, along with the attribute byte (which is retrieved from *cons->c_attr*), and *cons->c_rwords* (which is the index into the queue), *cons->c_column* (which keeps track of the column on the screen), and *tbuf*, the pointer into the buffer, are all incremented. This direct placement of characters into *cons->c_ramqueue* corresponds to the dashed line on the left side of Fig. 3-35. If needed, *out_char* is called (line 16082). It does all of the bookkeeping, and additionally calls *flush*, which does the final transfer to screen memory, when necessary.

The transfer from the user buffer to the local buffer to the queue is repeated as long as *tp->tty_outleft* indicates there are still characters to be transferred and the flag *tp->tty_inhibited* has not been raised. When the transfer stops, whether because the write operation is complete or because *tp->tty_inhibited* has been raised, *flush* is called again to transfer the last characters in the queue to screen memory. If the operation is complete (tested by seeing if *tp->tty_outleft* is zero), a reply message is sent by calling *tty_reply* lines 16096 and 16097).

In addition to calls to *cons_write* from *handle_events*, characters to be displayed are also sent to the console by *echo* and *rawecho* in the hardware-independent part of the terminal driver. If the console is the current output device, calls via the *tp->tty_echo* pointer are directed to the next function, *cons_echo* (line 16105). *Cons_echo* does all of its work by calling *out_char* and then *flush*. Input from the keyboard arrives character by character and the person doing the typing wants to see the echo with no perceptible delay, so putting characters into the output queue would be unsatisfactory.

Out_char (line 16119). does a test to see if an escape sequence is in progress, calling *parse_escape* and then returning immediately if so (lines 16124 to 16126). Otherwise, a switch is entered to check for special cases: null, backspace, the bell character, and so on. The handling of most of these is easy to follow. The linefeed and the tab are the most complicated, since they involve complicated changes to the position of the cursor on the screen and may require scrolling as well. The last test is for the *ESC* code. If it is found, the *cons->c_esc_state* flag is set (line 16181), and future calls to *out_char* are diverted to *parse_escape* until

the sequence is complete. At the end, the default is taken for printable characters. If the screen width has been exceeded, the screen may need to be scrolled, and *flush* is called. Before a character is placed in the output queue a test is made to see that the queue is not full, and *flush* is called if it is. Putting a character into the queue requires the same bookkeeping we saw earlier in *cons_write*.

The next function is *scroll_screen* (line 16205). *Scroll_screen* handles both scrolling up, the normal situation that must be dealt with whenever the bottom line on the screen is full, and scrolling down, which occurs when cursor positioning commands attempt to move the cursor beyond the top line of the screen. For each direction of scroll there are three possible methods. These are required to support different kinds of video cards.

We will look at the scrolling up case. To begin, *chars* is assigned the size of the screen minus one line. Softscrolling is accomplished by a single call to *vid_vid_copy* to move *chars* characters lower in memory, the size of the move being the number of characters in a line. *Vid_vid_copy* can wrap, that is, if asked to move a block of memory that overflows the upper end of the block assigned to the video display, it fetches the overflow portion from the low end of the memory block and moves it to an address higher than the part that is moved lower, treating the entire block as a circular array. The simplicity of the call hides a fairly slow operation, even though *vid_vid_copy* is an assembly language routine (defined in *drivers/tty/vidcopy.s*, not listed in Appendix B). This call requires the CPU to move 3840 bytes, which is a large job even in assembly language.

The softscroll method is never the default; the operator is supposed to select it only if hardware scrolling does not work or is not desired for some reason. One reason might be a desire to use the *screendump* command, either to save the screen memory in a file or to view the main console display when working from a remote terminal. When hardware scrolling is in effect, *screendump* is likely to give unexpected results, because the start of the screen memory is likely not to coincide with the start of the visible display.

On line 16226 the *wrap* variable is tested as the first part of a compound test. *Wrap* is true for older displays that can support hardware scrolling, and if the test fails, simple hardware scrolling occurs on line 16230, where the origin pointer used by the video controller chip, *cons->c_org*, is updated to point to the first character to be displayed at the upper-left corner of the display. If *wrap* is *FALSE*, the compound test continues with a test of whether the block to be moved up in the scroll operation overflows the bounds of the memory block designated for this console. If this is so, *vid_vid_copy* is called again to make a wrapped move of the block to the start of the console's allocated memory, and the origin pointer is updated. If there is no overlap, control passes to the simple hardware scrolling method always used by older video controllers. This consists of adjusting *cons->c_org* and then putting the new origin in the correct register of the controller chip. The call to do this is executed later, as is a call to blank the bottom line on the screen to achieve the "scrolling" effect.

The code for scrolling down is very similar to that for scrolling up. Finally, *mem_vid_copy* is called to blank out the line at the bottom (or top) addressed by *new_line*. Then *set_6845* is called to write the new origin from *cons->c_org* into the appropriate registers, and *flush* makes sure all changes become visible on the screen.

We have mentioned *flush* (line 16259) several times. It transfers the characters in the queue to the video memory using *mem_vid_copy*, updates some variables, and then makes sure the row and column numbers are reasonable, adjusting them if, for instance, an escape sequence has tried to move the cursor to a negative column position. Finally, a calculation of where the cursor ought to be is made and is compared with *cons->c_cur*. If they do not agree, and if the video memory that is currently being handled belongs to the current virtual console, a call to *set_6845* is made to set the correct value in the controller's cursor register.

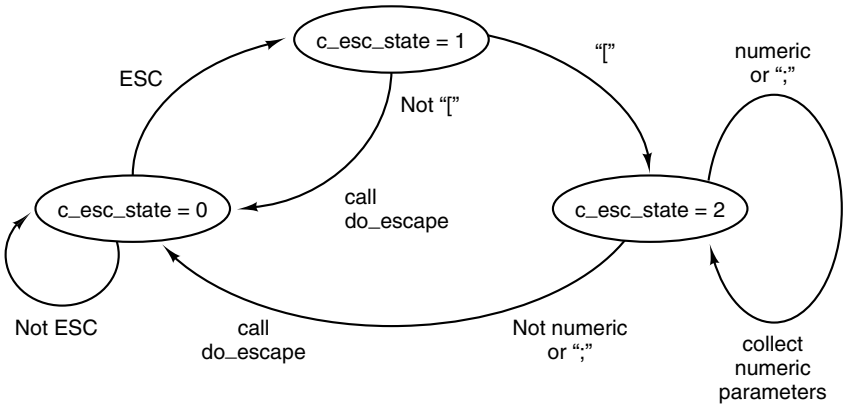

Figure 3-44. Finite state machine for processing escape sequences.

Figure 3-44 shows how escape sequence handling can be represented as a finite state machine. This is implemented by *parse_escape* (line 16293) which is called at the start of *out_char* if *cons->c_esc_state* is nonzero. An ESC itself is detected by *out_char* and makes *cons->c_esc_state* equal to 1. When the next character is received, *parse_escape* prepares for further processing by putting a "\0" in *cons->c_esc_intro*, a pointer to the start of the array of parameters, *cons->c_esc_parmv*[0] into *cons->c_esc_parmp*, and zeroes into the parameter array itself. Then the first character directly following the ESC is examined—valid values are either "[" or "M". In the first case the "[" is copied to *cons->c_esc_intro* and the state is advanced to 2. In the second case, *do_escape* is called to carry out the action, and the escape state is reset to zero. If the first character after the ESC is not one of the valid ones, it is ignored and succeeding characters are once again displayed normally.

When an ESC [sequence has been seen, the next character entered is processed by the escape state 2 code. There are three possibilities at this point. If the

character is a numeric character, its value is extracted and added to 10 times the existing value in the position currently pointed to by *cons->c_esc_parmp*, initially *cons->c_esc_parmv*[0] (which was initialized to zero). The escape state does not change. This makes it possible to enter a series of decimal digits and accumulate a large numeric parameter, although the maximum value currently recognized by MINIX 3 is 80, used by the sequence that moves the cursor to an arbitrary position (lines 16335 to 16337). If the character is a semicolon there is another parameter, so the pointer to the parameter string is advanced, allowing succeeding numeric values to be accumulated in the second parameter (lines 16339 to 16341). If *MAX_ESC_PARMS* were to be changed to allocate a larger array for the parameters, this code would not have to be altered to accumulate additional numeric values after entry of additional parameters. Finally, if the character is neither a numeric digit nor a semicolon, *do_escape* is called.

Do_escape (line 16352) is one of the longer functions in the MINIX 3 system source code, even though MINIX 3's complement of recognized escape sequences is relatively modest. For all its length, however, the code should be easy to follow. After an initial call to *flush* to make sure the video display is fully updated, there is a simple if choice, depending upon whether the character immediately following the ESC character was a special control sequence introducer or not. If not, there is only one valid action, moving the cursor up one line if the sequence was ESC M. Note that the test for the "M" is done within a switch with a default action, as a validity check and in anticipation of addition of other sequences that do not use the ESC [format. The action is typical of many escape sequences: the *cons->c_row* variable is inspected to determine if scrolling is required. If the cursor is already on row 0, a *SCROLL_DOWN* call is made to *scroll_screen*; otherwise the cursor is moved up one line. The latter is accomplished just by decrementing *cons->c_row* and then calling *flush*. If a control sequence introducer is found, the code following the else on line 16377 is taken. A test is made for "[", the only control sequence introducer currently recognized by MINIX 3. If the sequence is valid, the first parameter found in the escape sequence, or zero if no numeric parameter was entered, is assigned to *value* (line 16380). If the sequence is invalid, nothing happens except that the large switch that ensues (lines 16381 to 16586) is skipped and the escape state is reset to zero before returning from *do_escape*. In the more interesting case that the sequence is valid, the switch is entered. We will not discuss all the cases; we will just note several that are representative of the types of actions governed by escape sequences.

The first five sequences are generated, with no numeric arguments, by the four "arrow" keys and the Home key on the IBM PC keyboard. The first two, ESC [A and ESC [B, are similar to ESC M, except they can accept a numeric parameter and move up and down by more than one line, and they do not scroll the screen if the parameter specifies a move that exceeds the bounds of the screen. In such cases, *flush* catches requests to move out of bounds and limits the move to the last row or the first row, as appropriate. The next two sequences, ESC [C and

ESC [D, which move the cursor right and left, are similarly limited by *flush*. When generated by the "arrow" keys there is no numeric argument, and thus the default movement of one line or column occurs.

ESC [H can take two numeric parameters, for instance, ESC [20;60H. The parameters specify an absolute position rather than one relative to the current position and are converted from 1-based numbers to 0-based numbers for proper interpretation. The Home key generates the default (no parameters) sequence which moves the cursor to position (1, 1).

ESC [*s*J and ESC [*s*K clear a part of either the entire screen or the current line, depending upon the parameter that is entered. In each case a count of characters is calculated. For instance, for ESC [1J, *count* gets the number of characters from the start of the screen to the cursor position, and the count and a position parameter, *dst*, which may be the start of the screen, *cons->c_org*, or the current cursor position, *cons->c_cur*, are used as parameters to a call to *mem_vid_copy*. This procedure is called with a parameter that causes it to fill the specified region with the current background color.

The next four sequences insert and delete lines and spaces at the cursor position, and their actions do not require detailed explanation. The last case, ESC [*n*m (note the *n* represents a numeric parameter, but the "m" is a literal character) has its effect upon *cons->c_attr*, the attribute byte that is interleaved between the character codes when they are written to video memory.

The next function, *set_6845* (line 16594), is used whenever it is necessary to update the video controller chip. The 6845 has internal 16-bit registers that are programmed 8 bits at a time, and writing a single register requires four I/O port write operations. These are carried out by setting up an array (vector) of (port, value) pairs and invoking a sys_voutb kernel call to get the system task to do the I/O. Some of the registers of the 6845 video controller chip are shown in Fig. 3-45

Registers	Function
10 – 11	Cursor size
12 – 13	Start address for drawing screen
14 – 15	Cursor position

Figure 3-45. Some of the 6845's registers.

The next function is *get_6845* (line 16613), which returns the values of readable video controller registers. It also uses kernel calls to accomplish its job. It does not appear to be called from anywhere in the current MINIX 3 code, but it may be useful for future enhancements such as adding graphics support.

The *beep* function (line 16629) is called when a CTRL-G character must be output. It takes advantage of the built-in support provided by the PC for making sounds by sending a square wave to the speaker. The sound is initiated by more

of the kind of magic manipulation of I/O ports that only assembly language programmers can love. The more interesting part of the code is using the ability to set an alarm to turn off the beep. As a process with system privileges (i.e., a slot in the *priv* table), the terminal driver is allowed to set a timer using the library function *tmrs_settimers.* On line 16655 this is done, with the next function, *stop_beep*, specified as the function to run when the timer expires. This timer is put into the terminal task's own timer queue. The sys_setalarm kernel call that follows asks the system task to set a timer in the kernel. When that expires, a SYN_ALARM message is detected by the main loop of the terminal driver, *tty_task*, which calls *expire_timers* to deal with all timers belonging to the terminal driver, one of which is the one set by beep.

The next routine, *stop_beep* (line 16666), is the one whose address is put into the *tmr_func* field of the timer initiated by *beep.* It stops the beep after the designated time has elapsed and also resets the *beeping* flag. This prevents superfluous calls to the beep routine from having any effect.

Scr_init (line 16679) is called by *tty_init NR_CONS* times. Each time its argument is a pointer to a *tty* structure, one element of the *tty_table*. On lines 16693 and 16694 *line*, to be used as the index into the *cons_table* array, is calculated, tested for validity, and, if valid, used to initialize *cons*, the pointer to the current console table entry. At this point the *cons->c_tty* field can be initialized with the pointer to the main *tty* structure for the device, and, in turn, *tp->tty_priv* can be pointed to this device's *console_t* structure. Next, *kb_init* is called to initialize the keyboard, and then the pointers to device specific routines are set up, *tp->tty_devwrite* pointing to *cons_write*, *tp->tty_echo* pointing to *cons_echo*, and *tp->tty_ioctl* pointing to *cons_ioctl*. The I/O address of the base register of the CRT controller is fetched from the BIOS, the address and size of the video memory are determined on lines 16708 to 16731, and the *wrap* flag (used to determine how to scroll) is set according to the class of video controller in use. On line 16735 the segment descriptor for the video memory is initialized in the global descriptor table by the system task.

Next comes the initialization of virtual consoles. Each time *scr_init* is called, the argument is a different value of *tp*, and thus a different *line* and *cons* are used on lines 16750 to 16753 to provide each virtual console with its own share of the available video memory. Each screen is then blanked, ready to start, and finally console 0 is selected to be the first active one.

Several routines display output on behalf of the terminal driver itself, the kernel, or another system component. The first one, *kputc* (line 16775) just calls *putk*, a routine to output text a byte at a time, to be described below. This routine is here because the library routine that provides the *printf* function used within system components is written to be linked to a character printing routine with this name, but other functions in the terminal driver expect one named *putk*.

Do_new_kmess (line 16784) is used to print messages from the kernel. Actually, "messages'' is not the best word to use here; we do not mean messages as

used for interprocess communication. This function is for displaying text on the console to report information, warnings, or errors to the user.

The kernel needs a special mechanism to display information. It needs to be robust, too, so it can be used during startup, before all components of MINIX 3 are running, or during a panic, another time when major parts of the system may be unavailable. The kernel writes text into a circular character buffer, part of a structure that also contains pointers to the next byte to write and the size of the yet-to-be processed text. The kernel sends a *SYS_SIG* message to the terminal driver when there is new text, and *do_new_kmess* is called when the main loop in *tty_task* is running. When things are not going so smoothly, (i.e., when the system crashes) the *SYS_SIG* will be detected by the loop that includes a nonblocking read operation in *do_panic_dumps*, which we saw in *keyboard.c*, and *do_new_kmess* will be called from there. In either case, the kernel call sys_getkmessages retrieves a copy of the kernel structure, and the bytes are displayed, one by one, by passing them to *putk*, followed by a final call to *putk* with a null byte to force it to flush the output. A local static variable is used to keep track of the position in the buffer between messages.

Do_diagnostics (line 16823) has a function similar to that of *do_new_kmess*, but *do_diagnostics* is used to display messages from system processes, rather than the kernel. A *DIAGNOSTICS* message can be received either by the *tty_task* main loop or the loop in *do_panic_dumps*, and in either case a call is made to *do_diagnostics*. The message contains a pointer to a buffer in the calling process and a count of the size of the message. No local buffer is used; instead repeated sys_vircopy kernel calls are made to get the text one byte at a time. This protects the terminal driver; if something goes wrong and a process starts generates an excessive amount of output there is no buffer to overrun. The characters are output one by one by calling *putk*, followed by a null byte.

Putk (line 16850) can print characters on behalf of any code linked into the terminal driver, and is used by the functions just described to output text on behalf of the kernel or other system components. It just calls *out_char* for each non-null byte received, and then calls *flush* for the null byte at the end of the string.

The remaining routines in *console.c* are short and simple and we will review them quickly. *Toggle_scroll* (line 16869) does what its name says, it toggles the flag that determines whether hardware or software scrolling is used. It also displays a message at the current cursor position to identify the selected mode. *Cons_stop* (line 16881) reinitializes the console to the state that the boot monitor expects, prior to a shutdown or reboot. *Cons_org0* (line 16893) is used only when a change of scrolling mode is forced by the F3 key, or when preparing to shut down. *Select_console* (line 16917) selects a virtual console. It is called with the new index and calls *set_6845* twice to get the video controller to display the proper part of the video memory.

The next two routines are highly hardware-specific. *Con_loadfont* (line 16931) loads a font into a graphics adapter, in support of the ioctl *TIOCSFON*

operation. It calls *ga_program* (line 16971) to do a series of magical writes to an I/O port that cause the video adapter's font memory, which is normally not addressable by the CPU, to be visible. Then *phys_copy* is called to copy the font data to this area of memory, and another magic sequence is invoked to return the graphics adapter to its normal mode of operation.

The last function is *cons_ioctl* (line 16987). It performs only one function, setting the screen size, and is called only by *scr_init*, which uses values obtained from the BIOS. If there were a need for a real ioctl call to change the sizeMINIX 3screen code to provide the new dimensions would have to be written.

3.9 SUMMARY

Input/output is an important topic that is often neglected. A substantial fraction of any operating system is concerned with I/O. But I/O device drivers are often responsible for operating system problems. Drivers are often written by programmers working for device manufacturers. Conventional operating system designs usually require allowing drivers to have access to critical resources, such as interrupts, I/O ports, and memory belonging to other processes. The design of MINIX 3 isolates drivers as independent processes with limited privileges, so a bug in a driver cannot crash the entire system.

We started out by looking at I/O hardware, and the relation of I/O devices to I/O controllers, which are what the software has to deal with. Then we moved on to the four levels of I/O software: the interrupt routines, the device drivers, the device-independent I/O software, and the I/O libraries and spoolers that run in user space.

Then we examined the problem of deadlock and how it can be tackled. Deadlock occurs when a group of processes each have been granted exclusive access to some resources, and each one wants yet another resource that belongs to another process in the group. All of them are blocked and none will ever run again. Deadlock can be prevented by structuring the system so it can never occur, for example, by allowing a process to hold only one resource at any instant. It can also be avoided by examining each resource request to see if it leads to a situation in which deadlock is possible (an unsafe state) and denying or delaying those that lead to trouble.

Device drivers in MINIX 3 are implemented as independent processes running in user space. We have looked at the RAM disk driver, hard disk driver, and terminal driver. Each of these drivers has a main loop that gets requests and processes them, eventually sending back replies to report on what happened. Source code for the main loops and common functions of the RAM disk, hard disk, and floppy disk drivers is provided in a common driver library, but each driver is compiled and linked with its own copy of the library routines. Each