SECTION THREE

DEVICE DRIVER INTERFACE

3. DEVICE DRIVER INTERFACE

This section covers the details of device driver construction. The first three subsections describe the driver interface in terms of the environment in which drivers operate, the conventions that should be followed when implementing them, and the internal IDRIS resident routines which they can call. The remaining two subsections are tutorial in nature. They describe the construction of an ideal disk driver and an ideal terminal driver.

3.1. The Driver Environment

The purpose of a device driver is to make a real world device look, to the IDRIS utilities and applications, like a standard byte stream for sequential processing, or like an array of blocks for supporting random access filesystems. The real strength of IDRIS derives from this approach: devices look like files, and I/O is uniform. Since every disk known can hold an array of 512-byte blocks, some way or other, any disk (or other random access media) can hold a filesystem. All other real world devices can be interfaced as sources or sinks of byte streams.

The filenames which access the IDRIS drivers are, by convention, placed in the /dev directory. These are special files (block special and character special) which hold a device code instead of disk block pointers. This device code is passed to the driver when an access is made to the corresponding file. The high order 8-bits, known as the major number, serves as an index to a table of pointers to driver entry points. The low order 8-bits, known as the minor number, are interpreted by the driver only. Some drivers have meanings for all 8 bits, even though they only run one or two physical devices. The minor number is more often a simple unit number, and the major number usually refers to a single driver for a single controller board. Devices have names like /dev/rm0, /dev/com1, and /dev/rmy0.

The IDRIS resident always runs on behalf of some user process. It will not give control to another process while the process currently in control is executing within the resident. The only way to relinquish processing is to road block over some resource and willingly turn over control.

The exception to this rule is the processing of interrupts. Interrupts do not run on behalf of any user and are obliged to return control without blocking. An interrupt routine can make only a very restrictive set of calls on resident facilities.

The handling of interrupts themselves can be controlled via calls that enable or disable interrupts of various kinds. Often a category of interruption, such as an interrupt from any terminal, is disabled while the CPU executes some particularly sensitive piece of code relating to that category. This disabling happens both within the resident and within the device drivers themselves. It may happen under the auspices of some user or from within an interrupt service routine. The only obligation to be met is that interrupts eventually be re-enabled, and preferably soon.

3.1.1. Hooking Drivers into IDRIS

A device driver interfaces to the resident in three places:

1.) Device driver entry points are stored in a list embedded in the resident (in **main.c**).

2.) If interrupts are used, the hardware interrupt vector address for the device is embedded in the resident (in **a.s**).

3.) A device entry is placed in the filesystem (in the **/dev** directory).

All driver entry points are stored in two tables in the **main.c** module. The table **CDEVSW** is used for character special devices and the table **BDEVSW** is used for block special devices. Each of the members of a **CDEVSW** or **BDEVSW** are laid out to appear in memory as an array of structures, each structure holding all the addresses for one major device code.

Interrupts on the 8086 vector to specific trap addresses defined for each device. These trap addresses are set to vector to a little bit of assembly code in the **a.s** module. This code essentially converts a hardware interrupt into a C function call, with the C function called being the driver interrupt routine. The **device driver open** routine should call the **vector** driver support function to connect the hardware interrupt vector to the interface code in **a.s**. The inteface code in **a.s** should first call the **trapint** system routine to save the environment, then call the C function for the driver interrupt routine.

The **mkdev** utility program, described in Section IV of the IDRIS Users' Manual, is used to create the **/dev** filesystem entry for a device. When using **mkdev**, be careful in assigning major and minor numbers. The major number must match the location (index) of the **BDEVSW** or **CDEVSW** entry for the device. The minor numbers, although not used by the resident, must be compatible with the expectations of the driver. You can avoid mismatched table locations and major/minor numbers by using the IDRIS pseudo files **/dev/bnames** and **/dev/cnames** as input to the **mkdev** utility. These files are generated within the IDRIS resident, and can be read or piped. The 'data' contained in these files are the list of major/minor device names ordered according to the contents of **BDEVSW** (for **bnames**) and **CDEVSW** (for **cnames**).

3.2. Conventions

This section describes the header files and naming conventions used when writing or modifying a device driver.

3.2.1. Standard Header File

std.h defines the standard portable datatypes and a restricted set of "defines" for the C environment enhancement. Nearly all code written to interface to IDRIS first includes an instance of **std.h**.

3.2.2. Resident Header File

res.h defines the datatypes and a number of constants often used in the interface between user code and IDRIS resident services.

Here are some of the more important definitions in **res.h**. (It is a good idea to have a copy of **res.h** at hand while reading the following information):

BLOCK - the integer type used to hold a device block index. Its valid range is between [0, 65535].

DEV - the integer type used to pass around device major and minor numbers as indices to the table of device driver entry points. Note that this may be an index into either the character device table or the block device table, depending on context.

ERROR - the integer type used to hold a user error return code. Valid return codes are described in the Section II of this manual.

NODEV - the device major/minor number (of type DEV) used to indicate an unimplemented device. Its value is currently 0.

NOSIG - the address used to indicate that the signal handling for a particular signal has been turned off. Its value is currently 1.

ROOTINO - the inode number of the first inode on a disk. Its value is currently 1.

Also included in the **res.h** file are definitions for system calls, signals, error return codes, scheduler event priorities, and two macros needed for determining device major and minor numbers. The listing for these should be self-explanatory.

3.2.3. Block Special Device Header

bio.h defines the data types used by block special device drivers, such as disk drivers. Herewith are the principal definitions from **bio.h**. (It is a good idea to have a copy of **bio.h** at hand while reading the following information):

BUF - is the type of structure which governs the transfer of bytes for a particular block device to and from memory.

The first four members of **BUF** control the linkage to various internal lists and are maintained by calls to **enq**, **deq**, and other internal functions.

The next member is **b_flag**, which has several flag bits that define the types and values of data transfers. All other bits are used internally for system synchronization.

The member **b_dev**, contains the major and minor device indices.

The member **b_count** is the number of bytes to transfer. For filesystem transfers it will always be 512. For swap transfers it will be a multiple of 512, and for character special transfers it might be anything from 1 to 65535 (or larger in 32-bit machines).

The member **b_phys** contains the 32-bit physical address of the starting memory address for transfers.

The member **b_resid** contains the low 9 bits of the 32-bit seek address which is the seek offset within the block on the way into a device driver. This is guaranteed to be zero for filesystem and swap transfers. On the way out it is taken as the residual count of the bytes that were not transferred as requested. This will appear to the user as either a short read or a short write, depending on the actual system call. A short write is generally considered to be an error to user programs and IDRIS utilities.

The member **b_blkno** is a block address (in multiples of 512 bytes) used as an index into the device. It may have any value from 0 to 65535, although a device may not support block addresses above some limit.

DEVTAB - is the type of structure which controls the per-disk buffer cache.

The first four members are a union of the first four members of a **BUF** structure and are used similarly. Of interest, however, is **b_next**, which is zero if there are no buffers enqueued for action by the driver.

The fifth member, **d_stat**, is the device status. Typically it is nonzero, to indicate that the device is busy performing some operation. The device driver may use this entry for any purpose.

The last member, **d_nerr**, is used by the device driver to count the number of errors and subsequent re-try operations on a given block. It is up to the driver to decide how many re-trys to honor before giving up on a particular request.

Also included in the header are structure definitions that can be used as a guide for the construction of entries in the device tables actually defined in **a.s.**

BDEVSW - is the type of structure which holds the particular entry points that the resident will need for block special I/O requests to a device.

CDEVSW - is the type of structure which holds the particular entry points that the resident will need for character special I/O requests to a device.

Note that disk-type device drivers are not required to define alternate character special entry points. However, it is common to do so in order to provide formatting or sector-by-sector (raw) I/O.

3.2.4. Character Special Device Header File

cio.h defines the data types used by character special device drivers, such as terminals. Here are the principal definitions from **cio.h**. (It is a good idea to have a copy of **cio.h** at hand while reading the following information):

CLIST - is a 16-byte structure capable of holding 12 to 14 characters. These never need to be directly manipulated by device handlers.

CHQ - is the type of structure which controls the manipulation of CLIST structures. Of particular interest is the member **c_num**, used to indicate the number of characters currently in the queue.

TTY - is the type of structure which controls the front-end processing of characters between IDRIS and a terminal. It is dealt with through the set of routines named **settyp, ttin, ttread, ttrstart, ttset,** and **ttwrite.**

The first members of TTY are the queue controllers for input and output. Normal input can be line edited, typically using '\b' and Ctl-U. "Cooked" mode is seven-bit line-at-a-time input with input line editing, and carriage return/newline mapping. Output can be stopped and started using XON and XOFF. "Raw" mode is eight-bit character-at-a-time input with no interpretation.

The fourth member, **t_go**, is a pointer to the routine in the driver called to initiate output. It is called either by **ttin()** when echoing input under full duplex operation or by a write system call that is processed by **ttwrite()**.

The next member, **t_dev**, is the major and minor device code. Note that each terminal is indentified by a unique combination of major and minor device code. The major number is essentially a pointer to the driver itself, and is seldom used by the driver. The minor number is interpreted solely by the driver. It is typically a simple line number.

The member **t_stat** is the status of the terminal. This is maintained by both the driver and by standard IDRIS routines.

The member **t_open** is a count of the number of opens to the terminal.

The member **t_speeds** contains 2 bytes of speed information, the lower byte of which is used for input speed and flags. The higher byte is used for output

speed and flags. Most UARTS only support one speed for input and output, so only the low 4 bits are used in those cases.

The member **t_erase** is the character which erases the last input character (Default is '\b').

The member **t_kill** is the character which causes an input line to be deleted when in cooked mode.

The last member, **t_flag**, is used to determine the current processing behavior of **ttin** and associated routines. Actual bit positions are described under **gtty** in the Section II of this manual, since **t_speeds**, **t_erase**, **t_kill**, and **t_flag** are the data delivered to the user program via **gtty**, and settable via **stty**.

3.3. Resident Driver Support Functions

The IDRIS resident contains a set of data declarations and routines which device drivers should use to perform support functions. These are described in manual page format on the following pages.

NAME

biops - processor level for block devices

SYNOPSIS

```
IMPORT BITS biops;
```

FUNCTION

biops is the proper value for a call to **spl** to lockout disk drive interrupts during execution of interrupt-sensitive code that manipulates block special devices and block buffers.

EXAMPLE

```
IMPORT BITS biops;
BITS ps = spl(biops);

/* region protected from block I/O interrupts */
spl(ps);
```

SEE ALSO

spl, spl0, spl7, ttyps

NAME
brelse - release a buffer

SYNOPSIS
```
VOID brelse(pb)
     BUF *pb;
```

FUNCTION
brelse returns a buffer to the disk cache for reuse.

It may be called from an interrupt routine.

RETURNS
Nothing

EXAMPLE
```
diskini(getaddr (pb = getblk(NODEV)));
brelse(pb);
```

SEE ALSO
getblk

NAME

cmaptab - parity mapping table

SYNOPSIS

```
IMPORT TEXT cmaptab[128];
```

FUNCTION

cmaptab is a table of character remaps used mostly in tty drivers to convert parity. The test is simple: the expression (**cmaptab[i] & 0200**) is true when i is an ASCII value that has an odd number of bits. Additionally (**cmaptab[i] & 0100**) is true if the character does not necessitate cursor movement. This information is itself useful for memory-mapped video terminal drivers.

Other bit fields within **cmaptab** are magic from the standpoint of most implementors. They are used to select delays and translate uppercase to lowercase within the IDRIS resident.

EXAMPLE

To force odd parity:

```
if (c < 0200)
    c =| (cmaptab[c] & 0200) ^ 0200;
```

NAME

deq - remove buffer from queue

SYNOPSIS

```
BUF *deq(pd)
    BUF *pd;
```

FUNCTION

deq unlinks a BUF from the list of buffers at **pd** set up to handle block transfers to a given device.

deq may be called from an interrupt routine.

RETURNS

deq returns the buffer that was unlinked.

EXAMPLE

To mark the current floppy disk buffer as finished with:

```
iodone(deq(&rxtab));
```

SEE ALSO

enq

NAME

deqc - dequeue the next character to transmit

SYNOPSIS

```
TEXT deqc(queue)
    CHQ *queue;
```

FUNCTION

deqc is the routine called to remove a character from an input or output queue. It can be used by a device interrupt routine to retrieve the next character to be transmitted to a terminal controller for output. The argument, **queue**, is generally the address of the **t_outq** member of a TTY control structure as declared in **cio.h**.

deqc may be called from an interrupt handler routine.

RETURNS

deqc returns the dequeued character, or a negative number if the queue is empty.

EXAMPLE

```
/*  send next character to line printer
 */
VOID lpstart(pt)
    TTY *pt;
    {
    IMPORT LP11VEC *lpaddr;
    COUNT c;

    if (lpaddr->x_csr & READY && 0 <= (c = deqc(&pt->t_outq)))
        lpaddr->x_buf = c;
    }
```

SEE ALSO

ttin, ttread, ttstart, ttwrite, ttyps

NAME

deverr - print device error message

SYNOPSIS

```
VOID deverr(pb, pfmt, arg1, arg2, arg3, arg4)
    BUF *pb;
    TEXT *pfmt;
    BYTES arg1, arg2, arg3, arg4;
```

FUNCTION

deverr prints a device error message from information in the buffer controller pointed to by **pb**. **putfmt** is then called with the format string address **pfmt**, and up to four arguments. deverr then outputs a newline. The message is of the form:

```
error [reading/writing] block <blkno> on device <dev>:<rest>
```

deverr may be called from an interrupt routine.

EXAMPLE

```
if (5 <= ++idtab.d_nerr)
    deverr(pb, "failed, code %h", pv->id_errst);
```

SEE ALSO

putfmt, putlnm

NAME

dmajor - obtain major device index

SYNOPSIS

```
BYTES dmajor(dev)
    DEV dev;
```

FUNCTION

dmajor is a macro which obtains the major device number, or index into the system device tables, from a **DEV** code in a machine-independent fashion.

dmajor may be called from an interrupt routine.

RETURNS

dmajor returns the major device code in the range [0, 255].

EXAMPLE

```
rootmajor = dmajor(rootdev);
```

NAME

dminor - obtain minor device index

SYNOPSIS

```
BYTES dminor(dev)
    DEV dev;
```

FUNCTION

dminor obtains the minor device number, or index, in a machine-independent fashion. It is a macro defined in the header file **res.h** and used by device drivers that control multiple minor devices.

Note that IDRIS attaches no special meaning to the device minor code. It is passed to device driver entry points as data obtained from the character or block special inode involved. It is often interpreted within the device driver as a means of denoting either separate devices utilizing the same interface or separate flavors of the same device.

By convention, character special major/minor device code **1/0** is taken to mean the console terminal.

dminor may be invoked from an interrupt routine.

RETURNS

dminor returns the minor device code in the range [0, 255].

EXAMPLE

```
unit = dminor(pb->b_dev);
switch (unit & DENSITY)
    {
case SINGLEDEN:
    floppycmd(pb, 128);
    break;
case DOUBLEDEN:
    floppycmd(pb, 256);
    break;
case QUADDEN:
    floppycmd(pb, 512);
    break;
    }
```

NAME

enq - add buffer to list

SYNOPSIS

```
BUF *enq(pd, pb)
    BUF *pd, *pb;
```

FUNCTION

enq enqueues the buffer at pb to the end of the list at pd.

enq may be called from an interrupt routine.

RETURNS

enq returns pd.

EXAMPLE

```
/*  queue RL11 I/O
 */
VOID rlstrategy(pb)
    FAST BUF *pb;
    {
    IMPORT BITS biops;
    IMPORT DEVTAB rltab;
    IMPORT VOID rlstart();

    if (nblk[dminor(pb->b_dev) & 3] <= pb->b_blkno)
        {
        pb->b_flag |= B_ERROR;
        iodone(pb);
        }
    else
        {
        spl(biops);
        enq(&rltab, pb);
        if (!rltab.d_stat)
            rlstart();
        spl0();
        }
    }
```

SEE ALSO

deq

NAME

enqc - add a character to a queue

SYNOPSIS

```
BOOL enqc(c, queue)
    TEXT c;
    CHQ *queue;
```

FUNCTION

enqc is the routine called to add a character to an input or output queue. It can be used by device interrupt routines to communicate data to a base-level consumer. **ttin** uses **enqc**, for instance, to pass characters to **ttread**.

enqc may be called from an interrupt handler routine.

RETURNS

enqc returns YES if the system clist is full; otherwise it returns 0 if the queue can take the character.

EXAMPLE

The resident code to map tabs to spaces is:

```
case '\t':
    if (pt->t_flag & M_XTABS)
        do
            enqc(' ', &pt->t_outq);
            while (++pt->t_col & 03);
    break;
```

SEE ALSO

deqc, ttin, ttread

NAME

fetch - get a character from user buffer

SYNOPSIS

```
COUNT fetch()
```

FUNCTION

fetch will obtain the next character sent from user data space by a **write** or **stty** system call. The character is in the range [0, 255]. A -1 is returned when there are no more characters in the user buffer. The number of **fetch** calls is set by the **size** argument on a **write** system call, and to the full 2-byte or 4-byte count on an **stty** system call.

fetch may not be called from an interrupt routine.

RETURNS

fetch returns the character or a -1 on end of buffer.

EXAMPLE

To pick up the **stty** parameters:

```
/*  interpret stty and gtty in special form
 */
VOID mysgtty(dev, getfl)
    DEV dev;
    BOOL getfl;
    {
    IMPORT UTINY commbuf[];
    FAST COUNT i;

    if (getfl)
        for(i = 0; i < 6; ++i)
            setch(commbuf[i])j
    else
        for (i = 0; i < 6; ++i)
            commbuf[i] = fetch();
    }
```

SEE ALSO

setch

NAME

flush - clean out character I/O queues

SYNOPSIS

```
VOID flush(pt)
    TTY *pt;
```

FUNCTION

flush deletes all remaining characters in the queues maintained by the TTY structure at **pt**. It is used to discard the contents of a buffer when a device is first opened or to remove any typeahead (unread terminal characters) when **ttset** is called.

flush initializes the character queues for **pt**, turns off T_STOP and T_ESC from the **t_stat** field, and then does a **wakeup** on the output and raw input queues.

It cannot be called from an interrupt routine.

RETURNS

Nothing

SEE ALSO

wflush

NAME

getaddr - return buffer address

SYNOPSIS

```
TEXT *getaddr(pb)
    BUF *pb;
```

FUNCTION

getaddr returns **pb->b_addr**, the address in resident address space of the 512-byte buffer controlled by the BUF structure. It is used primarily for information hiding (ie. avoid the need for **bio.h**.

getaddr may be called from an interrupt routine.

RETURNS

getaddr returns the byte address of the 512-byte buffer controlled by the buffer controller pointed to by **pb**.

SEE ALSO

getblk

NAME

getblk - get an incore buffer

SYNOPSIS

```
BUF *getblk(dev, pbn)
    DEV dev;
    BLOCK pbn;
```

FUNCTION

getblk locates physical block **pbn** on the given device. The buffer will be locked, for exclusive use by the caller, until explicitly unlocked by a **brelse**, or **iodone** call. If the block is in the buffer cache, **B_VALID** will be set in the **b_flag** member of the structure pointed at by the return value.

If dev == NODEV, **pbn** is ignored, and **getblk** locates a free block. To prevent deadlock, no process is permitted to obtain more than one block at a time via **getblk**, unless special steps are taken (which are not documented here).

getblk may not be called from an interrupt routine.

RETURNS

getblk always returns a pointer to a BUF.

EXAMPLE

```
pbuf = getaddr(pb = getblk (NODEV));     /* use buffer */
brelse (pb);
```

SEE ALSO

brelse, getaddr

NAME

gsbyte - get a byte from system memory

SYNOPSIS

```
BYTES gsbyte(addr)
    UTINY *addr;
```

FUNCTION

gsbyte tries to read a byte at **addr** in the resident data space. Before reading the byte, it puts out a net to catch any bus failures caused by the read, so the system will not crash on an attempt to read memory that does not exist.

gsbyte may not be called from an interrupt routine.

RETURNS

gsbyte returns a **-1** if there was an access violation; otherwise, it returns the byte read as a number in the range [0, 255].

EXAMPLE

```
/*  let printer be opened by one process only
 */
VOID lpopen(dev)
    DEV dev;
    {
    IMPORT PIA *lpaddr;
    IMPORT TTY lp;
    IMPORT VOID lpstart();
    FAST PIA *pv = lpaddr;
    FAST TTY *pt = &lp;

    if (gsbyte(&lpaddr.lp_stat) < 0)  /* see if hardware is there */
        {
        uerror(ENXIO);
        return;
        }
    else if (pt->t_open)
        {
        uerror(EAGAIN);
        return;
        }
    pt->t_open = YES;
    pt->t_dev = dev;
    pt->t_go = &lpstart;
    pt->t_stat = T_OPEN|T_CARR;
    }
```

NAME
iodone - notify Idris of I/O completion

SYNOPSIS
```
BUF *iodone(pb)
    BUF *pb;
```

FUNCTION
iodone notifies IDRIS that any pending I/O on the BUF structure at **pb** is now complete.

If the bit **B_ERROR** in **b_flags** within **pb** is set, then the user will get an error return.

iodone may not be called from an interrupt routine.

RETURNS
iodone returns its argument, **pb**.

EXAMPLE
To notify all interested parties of a floppy disk error:

```
ioerror(pb, status.s_devcommand, status.s_status);
if (NRETRYS < ++fdctab.d_nerr)
    {
    fdctab.d_stat = NO;
    fdctab.d_nerr = 0;
    pb->b_flag |= B_ERROR;
    iodone(deq(&fdctab));
    }
fdcstart();
```

SEE ALSO
uerror

NAME

ioerror - print device error on console

SYNOPSIS

```
VOID ioerror(pb, a, b)
    BUF *pb;
    BYTES a, b;
```

FUNCTION

ioerror writes an error message to the console concerning the **BUF** structure at **pb** of the form:

error [reading/writing] block <blkno> on <device> <a> <b>

It is used to notify the operator that a device error has occurred. It should only be used to report failed I/O's, not "soft" recovered errors. The arguments **a** and **b** can be arbitrary, typically they represent controller error and status register values. The block number attempted is printed in decimal; **a** and **b** are printed in octal.

ioerror is written in terms of the more flexible **deverr** routine.

ioerror may be called from an interrupt routine.

RETURNS

Nothing

SEE ALSO

deverr, iotick, movbuf

BUGS

All the I/O is done to the console, and the processor suspends all other activity until the entire message is output.

NAME

iotick - account for I/O time

SYNOPSIS

```
VOID iotick(n)
    UCOUNT n;
```

FUNCTION

iotick subtracts n 1/60 second clock ticks from the currect process time slice, if the system clock is not running. If the system clock is running, **iotick** does nothing. It is used to provide pre-emptive scheduling in a multiprocess environment where there is no system clock. It is only used with non-interrupting I/O devices.

iotick may be called from an interrupt routine.

EXAMPLE

```
iotick(3);  /* Winchester I/O done */
```

NAME

movbuf - copy a buffer

SYNOPSIS

```
VOID movbuf(from, to, n)
    TEXT *from, *to;
    BYTES n;
```

FUNCTION

movbuf copies n bytes, in system data space. The source is from, the destination is to.

movbuf may be called from an interrupt routine.

EXAMPLE

```
movbuf(&tty[1].t_speeds, &tty[3].t_speeds, 6);
```

NAME

nodev - illegal device entry point

SYNOPSIS

VOID nodev()

FUNCTION

nodev will report the error **ENODEV** whenever it is called. It may be used in place of an **open, close, read, write, strategy,** or **setgetty** entry point to disallow use of the particular device function.

nodev may not be called from an interrupt routine.

EXAMPLE

To disallow reads:

```
/*  line printer read
 */
VOID lpread(dev)
    DEV dev;
    {
    nodev();
    }
```

SEE ALSO

nulldev

NAME

nulldev - innocuous device entry point

SYNOPSIS

VOID nulldev()

FUNCTION

nulldev does nothing, gracefully. It is a null function used to plug unused driver entry points. When used as a read-or-write device entry point, no characters are read or written, which is usually taken to mean end-of-file. As an **open**, **close** or **setgetty** entry point, it is a convenient no-op.

nulldev may not be called from an interrupt routine.

SEE ALSO

nodev

NAME

panic - send fatal message and die

SYNOPSIS

```
VOID panic(s)
    TEXT *s;
```

FUNCTION

panic prints the NUL-terminated fatal diagnostic message **s** to the console and hangs in the idle loop. It is used in dire circumstances where an immediate crash would be better than proliferating inconsistent data throughout the system.

A **sync** is performed by a **panic** call so that as much disk integrity as possible is maintained.

panic prints a message of the form:

```
IDRIS crash <s>
```

panic may be called from an interrupt routine.

RETURNS

panic will never return. The system will quiet down into idle and only interrupts will be serviced.

NAME

physio - set up character special I/O

SYNOPSIS

```
VOID physio(pstrat, dev, flags)
    VOID (*pstrat)();
    DEV dev;
    BITS flags;
```

FUNCTION

physio is used to set up an I/O operation directly to or from user space. No buffering is used. It is given a pointer to the **strategy** routine, and to the device code and flags that indicate the desired operation.

It will check the user space buffer for any machine dependent boundary segments, lock the user process into memory, allocate a **BUF** from a system pool, enqueue the I/O, and call the **strategy** routine. When the I/O completes, the above sequence is unwound.

If (flags & B_READ), the requested I/O is a read, otherwise it is a write. If (flags & **B_CTRL**), a non-data transfer is indicated, like tape rewind, so that the process need not be locked in memory.

physio may not be called from an interrupt routine.

RETURNS

Nothing. The desired operation is completed (i.e. it waits for **iodone**).

EXAMPLE

```
VOID tmwrite(dev);
    DEV dev;
    {
    IMPORT VOID tmstrategy();

    physio(&tmstrategy, dev, 0);
    }
```

NAME

putch - put a character to the console unbuffered

SYNOPSIS

```
VOID putch(c)
    TEXT c;
```

FUNCTION

putch sends its argument directly to the console terminal.

If **c** is a newline, then a carriage return, any necessary delay characters, and the newline are also sent.

putch is called by **putfmt**. **putch** should co-exist with interrupt-driven I/O to the console.

putch may be called from an interrupt routine.

RETURNS

Nothing

EXAMPLE

```
putch('!');   /* DEBUG - made it! */
```

SEE ALSO

putfmt

NAME

putdnm - print device name on console

SYNOPSIS

```
VOID putdnm(dev, chrflag)
    DEV dev;
    BOOL chrflag;
```

FUNCTION

putdnm uses the **dev** code to print out the device name on the console. If **chrflag** is non-zero, the name is a character device, otherwise it is a block device. The name is found by searching the name string supplied by driver for **dmajor(dev)**.

putdnm may be called from an interrupt routine.

EXAMPLE

```
putdnm(pb->b_dev, pb->b_flag & B_CHR);
putfmt(":    device is offline\n");
```

SEE ALSO

putfmt, deverr

NAME

putfmt - print formatted messages to console

SYNOPSIS

```
VOID putfmt(fmt, arg1, arg2, arg3, arg4)
    TEXT *fmt;
    BYTES arg1, arg2, arg3, arg4;
```

FUNCTION

putfmt converts a series of arguments (arg1, arg2, arg3, arg4) to text, which is output to the console under control of a format string at **fmt**. The format string consists of literal text to be output, interspersed with <field-specifier>s that determine how the arguments are to be interpreted and how they are to be converted for output.

A <field-specifier> takes the form:

%<field-code>

That is, a <field-specifier> consists of a literal % and is terminated by a <field-code>.

A <field-code> is one of the following:

h = hexadecimal (leading "0x")
o = octal (leading '0')
s = NUL-terminated string
u = unsigned decimal
i = signed decimal (leading '-' if < 0)

Any other character in the place of a <field-code> is taken as 'i', meaning signed decimal integer.

putfmt may be called from an interrupt routine.

RETURNS

Nothing.

EXAMPLE

```
putfmt("%i errors , pc = %h\n", nerrors, oldpc);
```

SEE ALSO

putch

BUGS

The system hangs until all characters are output to the console one at a time via **putch**.

NAME

setch - send a character to user buffer

SYNOPSIS

```
COUNT setch(c)
    COUNT c;
```

FUNCTION

setch places the character **c** in the next location to be filled in user data space, as specified by a **read** or **gtty** system call. The number of **setch** calls is set by the size argument of the read system call and to the full 2-byte or 4-byte count on a **gtty** system call.

setch may not be called from an interrupt routine.

RETURNS

setch returns -1 when the last requested character has been accepted. Otherwise it returns 0.

EXAMPLE

To read back a communication buffer via **gtty**:

```
/*  interpret stty and gtty in special form
 */
VOID mysgtty(dev, getfl)
    DEV dev;
    BOOL getfl;
    {
    IMPORT UTINY commbuf[];
    FAST COUNT cnt, i;

    if (getfl)
        for (i = 0; i < 6; ++i)
            if (setch(commbuf[i]))
                break;
    else
        for (i = 0; (cnt = fetch()) != -1 && i < 6; ++i)
            commbuf[i] = cnt & 0xff;
    }
```

SEE ALSO

fetch

NAME
settyp - offer to be controlling terminal

SYNOPSIS
```
VOID settyp(ttdev, set)
    DEV ttdev;
    BOOL set;
```

FUNCTION
settyp, when set is **YES**, connects **ttdev** to the current process as controlling terminal for all subsequent children and signal calls, if it does not already have a controlling terminal.

If set is **NO**, **settyp** will disassociate the terminal if it is the controlling terminal and the invoker is superuser.

This routine is called from all tty handlers - once on each open with a **YES**, and on the last close with a **NO**.

settyp may not be called from an interrupt routine.

RETURNS
Nothing

SEE ALSO
ttin, ttout

NAME

signal - send a kill signal

SYNOPSIS

```
VOID signal(ttdev, sig)
    DEV ttdev;
    TINY sig;
```

FUNCTION

signal sends the signal **sig** to all processes under control of **ttdev**.

The sendable signals are described under **kill** in Section II of this manual. In cooked mode, **ttin** will automatically send **SIGINT** on receipt of a **DEL** character and **SIGQIT** on receipt of an **FS** (034). Within the terminal driver, a framing error (break) is often mapped into **DEL** (and thus **SIGINT**).

signal may be called from an interrupt routine.

RETURNS

Nothing

EXAMPLE

```
/*  5-minute limit on this terminal
 */
VOID arm(pt)
    TTY pt;
    {
    timeout(&arm, pt, 5*60*60);
    if (pt->t_open)
        signal(pt->t_dev, SIGKIL);
    }
```

SEE ALSO

kill (in Section II of this manual)

NAME

sleep - wait for an event

SYNOPSIS

```
VOID sleep(chan, pri)
     TEXT *chan;
     COUNT pri;
```

FUNCTION

sleep suspends execution until a **wakeup** occurs on the same channel **chan**. The sleeping process, when awakened, will run with priority **pri**, which must be in the range [-127, 20].

Typical values for **pri** are PITY, for input from a TTY, POTY, for output to a TTY, and PBIO, for block I/O.

Note that processes with a non-negative priority (**PITY** or **POTY**)are interruptible and will return a system call error (**EINTR**) if sent a signal while sleeping.

A sleeping process with a negative priority (**PBIO**) will not be interrupted in this way.

sleep may NOT be called from an interrupt routine.

EXAMPLE

The resident code to wait for a BUF to come free:

```
spl7();
while (pb->b_flag & B_BUSY)
    {
    pb->b_flag |= B_WANT;
    sleep(pb, PBIO);
    }
spl0();
```

SEE ALSO

wakeup

NAME

spl - set arbitrary processor level

SYNOPSIS

```
BITS spl(npsw)
    BITS npsw;
```

FUNCTION

spl sets the program status word (processor level) to **npsw**. It can be used to set an arbitrary processor level or to re-establish an old one.

Note that the argument **npsw** should be a value acceptable to the native machine. It should be a value returned by **spl0** or **spl7**, or the value of **biops** or **ttyps**. Other hand-constructed values are dangerous and likely nonportable.

spl may be called for an interrupt routine provided the level is kept the same or raised.

RETURNS

spl returns the old processor level.

EXAMPLE

```
BITS ps = spl7();
/* non-interruptable code */
spl(ps);
```

SEE ALSO

biops, spl0, spl7, ttype

NAME

spl0 - enable all interrupts

SYNOPSIS

```
BITS spl0()
```

FUNCTION

spl0 enables all interrupts by lowering the processor level to 0.

spl0 may NOT be called from an interrupt routine.

RETURNS

spl0 returns the old processor level.

EXAMPLE

```
/*  queue FDC I/O
 */
VOID fdcstrat(pb)
    FAST BUF *pb;
    {
    IMPORT BITS biops;
    IMPORT DEVTAB fdctab;

    if (NBLK <= pb->b_blkno)
        {
        pb->b_flag |= B_ERROR;
        iodone(pb);
        }
    else
        {
        spl(biops);
        enq(&fdctab, pb);
        if (!fdctab.d_stat)
            fdcstart();
        spl0();
        }
    }
```

SEE ALSO

biops, spl, spl7, ttyps

NAME

spl7 - disable all interrupts

SYNOPSIS

```
BITS spl7()
```

FUNCTION

spl7 raises the processor level to the highest possible priority, effectively disabling all interrupts.

It is used when calling **timeout** to protect the running process from clock interrupts.

spl7 may be called from an interrupt routine.

RETURNS

spl7 returns the old processor level.

EXAMPLE

```
/*  start I/O on FDC
 */
VOID fdcstart()
    {
    IMPORT BOOL daemon;
    IMPORT COUNT timeleft;
    IMPORT DEVTAB fdctab;
    IMPORT UTINY *fdcbase;
    IMPORT VOID fdctmo();
    FAST BITS ps;
    FAST BUF *pb;
    COM cmd;

    if (pb = fdctab.d_next)
        {
        cmd.c_ctype = (pb->b_flag & B_READ) ? READ : WRITE;
        cmd.c_bcount = pb->b_count
        cmd.c_addr = pb->b_addr;
        cmd.c_secnum = pb->b_blkno
        if (!putpkt(&cmd, fdcbase))
            putfmt("can't write packet\n");
        fdctab.d_stat = YES;
        ps = spl7();
        timeleft = 5;
        if (!daemon)
            {
            timeout(&fdctmo, 0, 60);
            daemon = YES;
            }
        spl (ps);
        }
    }
```

SEE ALSO

biops, spl, spl0, ttyps

NAME

timeout - call function from clock interrupt after specified time

SYNOPSIS

```
VOID timeout(pfn, arg, t)
    VOID (*pfn)();
    TEXT *arg;
    COUNT t;
```

FUNCTION

timeout schedules a call to the function at **pfn** with the single argument **arg** in **t** 60ths of a second. It can be used to check for changes in data set status, implement transmission delays, and signal device controller overdue, to name but a few uses.

The function is called at the clock interrupt level, with the same restrictions as a typical device interrupt handler. If **t** is 0, the function is called immediately.

The function is called only once per **timeout** request. The call will be deferred until the clock tick has not interrupted an interrupt. Up to 20 events can be registered to **timeout**.

timeout may be called from an interrupt routine.

RETURNS

timeout returns immediately, with no meaningful value.

EXAMPLE

```
/*  timeout FDC interrupt
 */
VOID fdctmo()
    {
    IMPORT BOOL daemon;
    IMPORT COUNT timeleft;

    daemon = NO;
    if (!timeleft)
        ;
    else if (!--timeleft)
        {
        putfmt("fdc timed out\n");
        fdcstart();
        }
    else
        {
        daemon = YES;
        timeout(&fdctmo, 0, 60);
        }
    }
```

BUGS

The **timeout** cannot be cancelled. The system will crash if there are too many active calls, since the table space is limited.

NAME

ttin - put a character on the input list

SYNOPSIS

```
VOID ttin(c, pt)
    COUNT c;
    TTY *pt;
```

FUNCTION

ttin puts the character **c** on the input queue associated with the TTY structure at **pt**.

On the way, it handles processing of character echoing, line editing, carriage return/newline mapping, uppercase-to-lowercase translation, and parity bit stripping for ASCII. Only the low seven bits are preserved; the eighth (parity) bit is set to zero. It will send an interrupt signal if it is sent 0x7f (DEL), and will issue a **wakeup** at a delimiter such as '\n' (LF) or ctl-D (EOT) in cooked mode.

When in_xoff characters have been passed, an **X-OFF** (DC3) character is sent. When **in_max** characters have been passed, all input is flushed. When **in_lim** is reached, a ctl-D is simulated. Only **in_max** is checked in raw mode.

Raw mode (pt->t_flag & M_RAW) is optimized to bypass much of this checking. It may issue a **wakeup** on every character if necessary; and it passes all characters unmodified.

ttin may be called from within an interrupt routine.

RETURNS

Nothing

EXAMPLE

```
csr = pv->r_csr;
if (csr & DONE)
    if ((buf = pv->r_buf) & FRAMERR)
        ttin(CINTR, pt);
    else
        ttin(buf, pt);
```

SEE ALSO

ttout, ttread, ttset, ttwrite

NAME

ttread - transfer characters to user buffer

SYNOPSIS

```
VOID ttread(pt)
    TTY *pt;
```

FUNCTION

ttread copies input lines into the buffer in user data space. If TTY is in cooked mode, no characters are available until at least one delimiter (ctl-D or newline) has been enqueued; characters are copied until a delimiter is reached. In raw mode, all available characters, up to the size requested, are copied. **ttread** need only be invoked once per read request. The invoking process may be suspended until at least one character can be sent, or, in cooked mode, until a ctl-D is encountered.

ttread may not be called from within an interrupt routine.

EXAMPLE

```
/* complete a read request for DL11 characters
 */
VOID dlread(dev)
    DEV dev;
    {
    IMPORT TTY dl11[NDL11];

    ttread(&dl11[dminor(dev)]);
    }
```

SEE ALSO

ttin, ttset, ttrstart, ttwrite

NAME

ttrstart - restart terminal after delays

SYNOPSIS

```
VOID ttrstart(pt)
    TTY *pt;
```

FUNCTION

ttrstart is a routine suitable for a call to **timeout** that will restart the I/O on the given terminal after a transmitter timeout delay. It calls the routine pointed to by **pt->t_go**.

ttrstart may be called from within an interrupt routine.

EXAMPLE

```
/*  start transmitter
 */
VOID dlstart(pt)
    FAST TTY *pt;
    {
    IMPORT DLVEC *dladdr[];
    IMPORT TEXT cmaptab[];
    IMPORT VOID ttrstart();
    FAST COUNT c;
    FAST DLVEC *pv = dladdr[pt->t_dev.d_minor];

    if (pt->t_stat & (T_STOP|T_WCLOS|T_TIMER) || !(pv->x_csr & DONE)
        || (c = deqc(&pt->t_outq)) < 0)
        ;
    else if (pt->t_flag & M_RAW)
        {
        pv->x_csr =| IENABLE;
        pv->x_buf = c;
        }
    else if (c < 0200)
        {
        pv->x_csr =| IENABLE;
        pv->x_buf = cmaptab[c] & 0200 | c;      /*  even parity  */
        }
    else
        {
        timeout(&ttrstart, pt, c & 0177);
        pt->t_stat =| T_TIMER;
        }
    }
```

SEE ALSO

timeout

NAME

ttset - complete stty processing

SYNOPSIS

```
BOOL ttset(pt, getfl)
    TTY *pt;
    BOOL getfl;
```

FUNCTION

ttset merges the data obtained from an **stty** system call into the TTY structure at **pt** so that the next call to **ttin**, **ttread**, or **ttwrite** will have the desired effect. **ttin** also sets up the flags in **t_speeds** indicating input ready and output ready. It clears input lost and break received.

If **getfl** is YES, the current values are copied out to the user data space.

The only parameters of a typical serial port that are not manipulated through a call to **ttset** are the speeds and modem control signals. These are best set after each call to **ttset** that has a NO as second parameter.

ttset may not be called from within an interrupt routine.

RETURNS

ttset returns its second argument, **getfl**.

EXAMPLE

```
/* process an stty call for a 2-speed modem
 */
VOID mdsgtty(dev, getfl)
    DEV dev;
    BOOL getfl;
    {
    IMPORT MDVEC *modemaddr[];
    IMPORT TTY modems[];
    FAST BITS speed;
    FAST MDVEC *pv = modemaddr[dminor(dev)];
    FAST TTY *pt = &modems[dminor(dev)];

    if (ttset(pt, getfl))
        return;
    if (speed = pt->t_speeds & S_ISPEED)
        pv->x_highspeedbit = speed & 1;
    else
        pv->x_hangupbit = YES;
    }
```

SEE ALSO

setch, fetch, ttin, ttread, ttwrite

NAME

ttwrite - start transmission from user buffer

SYNOPSIS

```
VOID ttwrite(pt)
    TTY *pt;
```

FUNCTION

ttwrite enqueues the characters from the user data space buffer for transmission, and invokes the transmitter routine (pt->t_go). **ttwrite** need be invoked only once per write request. The invoking process may be suspended until all characters have been enqueued for output.

ttwrite may not be called from within an interrupt routine.

EXAMPLE

```
/* complete a write request for DL11 characters
 */
VOID dlwrite(dev)
    DEV dev;
    {
    IMPORT TTY dl11[NDL11];

    ttwrite(&dl11[dminor(dev)]);
    }
```

SEE ALSO

ttin, ttset, ttread, ttrstart

NAME

ttyps - processor level for terminal devices

SYNOPSIS

```
IMPORT BITS ttyps;
```

FUNCTION

ttyps is the proper value for a call to **spl** to lockout terminal interrupts during execution of interrupt sensitive code that manipulates character queues and TTY structures.

EXAMPLE

```
IMPORT BITS ttyps;
BITS ps = spl(ttyps);

/* no terminal interrupts allowed */
spl(ps);
```

SEE ALSO

biops, spl spl0, spl7

NAME

uerror - set or test for user error

SYNOPSIS

```
COUNT uerror(err)
    COUNT err;
```

FUNCTION

uerror records an error, if err is nonzero, to be returned to the user or process invoking the current system call. It can be used to set or test the current error return status for a system call.

uerror may not be called from within an interrupt handler.

RETURNS

uerror returns the current user error return status, possibly after setting it to err.

EXAMPLE

```
if (!uerror(0))
    uerror(EAGAIN);
```

SEE ALSO

iodone

NAME

vector - redirect 8086 interrupt

SYNOPSIS

```
VOID vector(vno, vip, vcs)
    BYTES vno, vip, vcs;
```

FUNCTION

vector redirects an 8086 interrupt and saves the original setting. If it hasn't already been saved, the current interrupt vector instruction pointer (ip) and code section (cs) are saved in an internal table for restarting should IDRIS terminate operation. **vip** is then stored in absolute location **vno*4**, and **vcs** is stored in **vno*4+2**. **vector** may be called more than once for any interrupt location.

For 8086 device drivers, **vno** is a number in the range [0, 255], **vip** is the address of an assembly language routine in **a.s**, and **vcs** is the contents of the data segment register when the resident is running.

vector may not be called from within an interrupt routine.

EXAMPLE

```
IMPORT BYTES dseg;
IMPORT VOID aca0();

vector(12,&aca0, dseg);
```

NAME

wakeup - post event for all waiters

SYNOPSIS

```
VOID wakeup(chan)
    TEXT *chan;
```

FUNCTION

wakeup looks for processes sleeping on channel **chan** and puts those that match into the RUNNING state with the priority specified by the **sleep** call. If no processes are sleeping on **chan**, the event goes unnoticed. **chan** is generally some agreed-upon address within the resident data space.

wakeup may be called from an interrupt handler.

EXAMPLE

The resident code called eventually by **iodone**:

```
/* wakeup sleeping processes in a buffer
 */
VOID wakeb(pb)
    FAST BUF *pb;
    {
    if (pb->b_flag & B_WANT)
        {
        pb->b_flag &= ~B_WANT;
        wakeup(pb);
        }
    }
```

SEE ALSO

sleep

NAME
wflush - wait for tty I/O to drain

SYNOPSIS
```
VOID wflush(pt)
    TTY *pt;
```

FUNCTION
wflush will flush the queues associated with the TTY structure at **pt** after waiting for all pending output on **pt** to come to completion. Execution resumes when **pt** is empty.

wflush may NOT be called from within an interrupt routine.

EXAMPLE
```
/*  close a CN
 */
VOID cnclose(dev)
    DEV dev;
    {
    IMPORT ACIA *cnaddr[];
    IMPORT BITS ttyps;
    IMPORT TTY cn[];
    IMPORT UTINY cnstat[];
    FAST UTINY devm = dminor(dev);
    FAST TTY *pt = &cn[devm];

    if (--pt->t_open)
        return;
    wflush(pt);
    spl (ttyps);
    pt->t_stat = 0;
    cnstat[devm] = C_BASE;
    cnaddr[devm]->a_stat = cnstat[devm];
    spl0();
    settyp(dev, NO);
    }
```

SEE ALSO
flush

3.4. Diskdriver Tutorial

This tutorial covers the common aspects of disk driver construction. In writing it, we have made some assumptions. First, the disk drive we are interfacing to is driven by a controller that can handle up to IDMAX drives, counting from zero. (IDMAX is user-defined.) Second, the disk drive is a new, fancy, smart, interrupt-driven model with 256 byte sectors. Obviously, if the drive you want to interface has larger or smaller sectors, you will want to change the algorithm slightly. The third and easiest assumption is that each logical drive maps to a single physical drive and that the device minor code selects the logical drive.

It is possible to map in other ways. Mapping multiple logical drives onto one physical drive is often just a matter of adding a cylinder offset to each seek position from a table indexed by the minor device number. In fact, this is what must be done to map large disks (greater than 32Mb). The disk is logically broken into chunks smaller than 32Mb, preferably into a size that is easily backed up.

3.4.1. Establishing Driver Entry Points

The following synopsis identifies the structure used by IDRIS to talk to disk-type devices. It lists each of the entry points into the device driver that IDRIS will use to do disk I/O.

```
typdef struct {
    VOID (*d_open)();
    VOID (*d_close)();
    VOID (*d_strat)();
    DEVTAB *d_tab;
    TEXT *d_bname;
    } BDEVSW;
```

The driver entry points are made known via two addresses. One is plugged into **blkdevs** and the other into **chrdevs**. Both tables are in **main.c**.

The code to set up the driver entry points is:

```
/* IDEAL DISK DRIVER
 * Copyright (c) 1984 by Ideal Disk Corporation
 * a wishful subsidiary of Whitesmiths, Ltd.
 */
#include <std.h>
#include <res.h>
#include <bio.h>
```

```
BDEVSW idbdev = {
    &idopen, &idclose, &idstrat, &idtab, nmbid};
    }

CDEVSW idcdev = {
    &idopen, &idclose, &idread, &idwrite, &nodev, nmcid};

LOCAL TEXT nmbid = {"id0 id1"};

LOCAL TEXT nmbid = {"rid0 rid1"};

LOCAL TEXT DEVTAB idtab = {0};
```

The interrupt routine address is the only other external. It is referenced by **a.s**, the interrupt module.

3.4.2. Block I/O Open

The easiest routine to write is the open routine. In the example below, and in those that follow, each line is numbered for easy reference. For a first try the code would look like:

```
1:    /*  open an id-type device
2:     */
3:    VOID idopen(dev, wrflag)
4:        DEV dev;
5:        BOOL wrflag;
6:        {
7:        /* initdrive(dminor(dev)); */
8:        }
```

Line 3 defines the entry point that IDRIS will use when the device is accessed for the first time. This is done once each time the device is mounted and once each time a user program opens the corresponding node from **/dev**.

The argument **dev** in line 4 is a concatenation of the device major and minor indexes. It is provided by IDRIS so that device drivers may have a way of determining what flavor of device is desired. For now it may be safely ignored.

The argument **wrflag** is YES if the disk is opened for update, otherwise it is set to NO (read-only).

Note that in line 7 the code to initialize the device was commented out. Most modern hardware needs no initialization. The power-on sequence suffices.

With this kind of simplicity, we could be tempted to do more. It is here that many of the validity checks for a device can be made. In this case we will be ambitious and force the open to fail if the drive is offline when the open is attempted. This code might look like:

```
 8:    /*  open an id-type device (Ambitious)
 9:     */
10:    LOCAL VOID idopen(dev)
11:        DEV dev;
12:        {
13:        IMPORT IDVEC *contaddr;
14:
15:        if (IDMAX < dminor(dev))
16:            uerror(ENXIO);
17:        else if (contaddr->id_online & (1 << dminor(dev)))
18:            uerror(EAGAIN);
19:        /*
20:        else
21:            initdrive(dminor(dev));
22:         */
23:        }
```

Here there are a few new assumptions:

- There exists a datatype, called IDVEC for convenience. It is a structure with various members that look like, and come in the same order as, the device registers on the controller for an id-type device. If lines 8 through 23 were part of a device driver listing, you would have already seen its declaration at the top of the file. In line 17 we notice it has a member called **id_online**. The member seems to have one bit for each drive and the bit is 1 if the drive is offline. Your machine may not have the drive online feature, but it is only that -- a feature.

- The variable **contaddr** is a pointer to an IDVEC. This is probably initialized to the address of the first device controller register. On a system that has multiple controllers, this parameter might be an array entry indexed by some combination of bits from the minor device code.

- The error code for an offline device can be different from a nonexistent device. This works in many cases because programs and utilities often check only whether or not an error has occurred, and not what kind. Applications programs that need to know about specific types of errors can still make the distinction themselves this way.

If you want to boot from this device, IDRIS automatically mounts the root device at startup and thus calls the open routine. Thus, there is no need to write special code for this function.

3.4.3. Block Device Close

The next easiest routine to implement is the close routine.

Here is the code:

```
24:    /*  close an id-type device
25:     */
26:    LOCAL VOID idclose(dev)
27:        DEV dev;
28:        {
```

```
29:         }
```

Too simple? Again, many devices need no extra attention when they are being closed. Even the flushing of unwritten blocks is taken care of before the close routine is called.

3.4.4. The Device Name

At this point it is best to start considering a name for your device. The manufacturer might call it an "ID-7" for idealized disk, model seven. The name string is a space-separated list of names for each minor number defined. Spaces in the string are significant and are used to skip over unused minor numbers. The string is read by IDRIS utilities via **/dev/bnames**. The commands:

```
#cd /dev
#mkdev -i -b0 < bnames
```

will establish entries in the **/dev** directory for all block-special devices, using these name strings. The name string is used by the IDRIS library function **getdn()**, and is meaningful to all IDRIS utilities that care about block device names. The code to initialize it could be:

```
30:     LOCAL TEXT nmbid[] = {"id0 id1"};
```

Line 30 defines two minor names, corresponding to IDMAX equal to 2.

3.4.5. Strategy for Block I/O

Bytes to be transferred to and from memory are managed by a BUF structure. Most of these transfer requests come from the resident as a multiple of 512 bytes. The rest come from raw device requests on the part of the user. In any case, all the requests eventually wind up calling a routine that designates a strategy for fulfilling the requests one at a time.

There is one strategy routine for each major device. IDRIS passes to the strategy routine a pointer to a BUF structure. It is up to the strategy routine, one of its descendants, or the interrupt routines to call **iodone** with that pointer as an argument. If the strategy routine returns back into the resident without calling **iodone**, the requesting program will road-block, unless the request is for a read-ahead block.

Associated with each strategy routine is a static structure called a DEVTAB. It controls a list of BUF structures through calls to **enq** and **deq** from the strategy routine. The head of that list is the current BUF structure being worked on by either the strategy routine or a descendant. The tail of the list is the latest BUF structure sent by IDRIS to the strategy routine.

With this simple set of transactions, let's write the final pieces of a block device driver.

First, we will need a start routine. This is the routine that will issue the necessary I/O start by either the strategy routine or a descendant.

```
31:     /*  start I/O on id drive
32:      */
33:     LOCAL VOID idstart()
34:         {
35:         IMPORT DEVTAB idtab;
36:         IMPORT IDVEC *contaddr;
37:         FAST BUF *pb;
38:         FAST IDVEC *pv = *contaddr;
39:
40:         if (pb = idtab.d_next)
41:             {
42:             idtab.d_stat = YES;
43:             pv->id_drvsel = dminor(pb->b_dev);
44:             pv->id_secsiz = 256;
45:             pv->id_secnum = (pb->b_blkno << 1) | (pb->b_resid >> 8);
46:             pv->id_secoff = pb->b_resid & 0xff;
47:             pv->id_memadd = pb->b_phys & 0xffff;
48:             pv->id_xtradd = pb->b_phys >> 16;
49:             pv->id_bytcnt =
50:                 -pb->b_count;/* incrementing counter */
51:             pv->id_cmd = (pb->b_flag & B_READ) ? RDCMD : WRTCMD;
52:             pv->id_godoit = YES;
53:             }
54:         }
```

Line 35 brings in the DEVTAB for this device. Line 36 brings in what is necessary to talk to the device controller. Line 38 is an optimization. It declares a register variable to be used as a pointer to the beginning of the device controller registers. This is done on the assumption that these device registers will be accessed frequently.

For the executable part we actually start at line 40, whose job is determining whether or not there is more work to do. This makes the **idstart** routine robust in case it gets called accidentally via a spurious interrupt. Line 42 sets a bit indicating that the drive is busy executing a command. Lines 43 through 46 calculate the actual read-or-write position on the disk. Lines 47 and 48 determine the memory location that the read-or-write is to go to or come from. Lines 49 and 50 tell the device how many bytes to transfer, and line 51 indicates to the device whether to do a read or a write. Finally, line 52 flips the bit that tells the device to begin the transfer.

Considering the complexity of the typical disk drive, this is actually simple. The routine was simple because our imaginary device was forgiving. The controller can address all the physical memory of the machine.

It crosses sector boundaries automatically and does the actual shoveling of data into memory, using DMA. The device also positions the heads by itself, needing only the sector number to start from. This also means that track and sector interleaving don't need to be computed. The disk spirals inward on an internal hardware algorithm. However, the device you interface may need interleaving, especially if you plan to read disks written by someone else.

Also note that the device allows the transfer to start from someplace other than the beginning of a sector, the offset within the sector being specified by **b_resid** on line 46. It is a rare disk that is this smart - IDRIS neither requires nor expects disk transfers to boundaries other than 512 bytes. Additionally, the LOCAL definition on line 33 isn't actually necessary; it is a reminder that the routine needs to be known only by the device driver itself. IDRIS will never call it directly, and the symbol table of global definitions doesn't need to be cluttered with it.

The next routine we will need is one that will handle the interrupts when the operation completes.

```
55:     /*  field all interrupts from id controller
56:      */
57:     VOID idintr()
58:         {
59:         IMPORT DEVTAB idtab;
60:         IMPORT IDVEC *contaddr;
61:         FAST BUF *pb = idtab.d_next;
62:         FAST IDEVC *pv = *contaddr;
63:
64:         if (!idtab.d_stat) /* spurious interrupt */
65:             return;
66:         idtab.d_stat = NO;
67:         if (pv->id_cmderr)
68:             {
69:             ioerror(pb, pv->id_error, pv->id_cmd)
70:             if (++idtab.d_nerr <= 10)
71:                 {
72:                 idstart();
73:                 return;
74:                 }
75:             pb->b_flag |= B_ERROR;
76:             }
77:         pb->b_resid = -pv->id_bytcnt;
78:         idtab.d_nerr = 0;
79:         iodone(deq(&idtab)));
80:         idstart();
81:         }
```

This routine does a lot of work. Mostly it handles errors and re-tries. Lines 64 through 66 protect the code from being invoked if the controller wasn't authentically busy. We set ourselves up to catch this one on line 42 above. Lines 67 through 70 attempt a re-try if the device signals an error in its last transfer.

Up to 10 re-tries are made, and if those aren't enough, line 75 marks the buffer so that the process issuing the request gets an error return. Line 77 sends back the real amount left unsatisfied, and line 78 resets the error status of the device so that the next transfer starts with a "clean slate." Line 79 notifies the system that the transfer is over and that any processes suspended while waiting for the transfer may now continue.

Note that **idintr** calls the start routine just before it finishes on line 80. This gives the device a "kick" in case there is more work to do. This work could have been enqueued while the device was off doing the last transfer. Also note that there is no setting of the processor interrupt level. The interrupt routine should never be re-entered.

The strategy routine sews it all together. We will also declare storage for a DEVTAB for the strategy routine, the start routine, and the interrupt routine to use.

```
 82:    LOCAL DEVTAB idtab = {0};
 83:
 84:    /*  queue I/O in id drive (Interrupt Version)
 85:     */
 86:    VOID idstrat(pb)
 87:        FAST BUF *pb;
 88:        {
 89:        IMPORT BITS biops;
 90:        IMPORT DEVTAB idtab;
 91:
 92:        if (MAXBLOCKS <= pb->b_blkno)
 93:            {
 94:            pb->b_flag |= B_ERROR;
 95:            iodone(pb);
 96:            }
 97:        else
 98:            {
 99:            spl(biops);
100:            enq(&idtab, pb);
101:            if (!idtab.d_stat)
102:                idstart();
103:            spl0();
104:            }
105:        }
```

Line 92 checks each incoming request to see that it refers to a valid area on the disk. MAXBLOCKS is the upper limit of the number of blocks that this device can hold. Since a block is a multiple of 512 bytes, a sample 10 megabyte drive could hold 20480 blocks (roughly). Lines 94 and 95 reject the request if it is invalid.

Line 99 locks out further interrupts from all disk-type devices so that no race conditions can occur.

Lines 101 and 102 start the device going if it is quiescent, and line 103 re-enables all interrupts so that other devices, which may have completed their commands, may now be serviced.

3.4.6. Raw Device Support

Often straight block-at-a-time transfers are not enough. Tape drives having block I/O capability only are inflexible because they cannot read or write tapes having differing blocking factors. Disk drives with removable media suffer a similar drawback. Access to the sector or record level is a pos-

sibility with the character special, or raw, device. Character special transfers allow processes to specify the exact number of characters to be transferred. Alignment and size restrictions depend purely upon what the device controller is capable of. The following synopsis lists the entry points needed for a character special interface:

```
typdef struct {
  VOID (*d_open)();
  VOID (*d_close)();
  VOID (*d_read)();
  VOID (*d_write)();
  VOID (*d_sgtty)();
  TEXT *d_cname;
  } CDEVSW;
```

Note that the code already written to satisfy block device opens and closes can be re-used for character opens and closes. The code entered so far can deal with transfers on unusual boundaries; it is just that IDRIS never asks for them on its own. IDRIS calls the strategy routine directly when doing swap or filesystem transfers. Filesystem transfers can be distinguished from swaps because they are destined for "system space". On most machines this means that the **b_phys** address is less than the value of the variable **systop** when used as a pointer (for those of you who don't have the benefit of DMA). Processes doing character special I/O have read/write routines which generally call **physio**, which sets the **B_CHR** bit in the **b_flags** field of the BUF structure so that such transfers can be distinguished from the norm.

The first addition we will make is to declare a raw device name.

```
106:    LOCAL TEXT ridname[ ] {"rid0 rid1"};
```

It is permissible to cause an error if character special transfers do not obey device restrictions. If the device must do its DMA to a particular boundary, it is okay to signal an error if this is not obeyed. Block special requests, however, should be the first to be supported. They are the least demanding of all requests; they call the strategy routine directly, are always sent to a block boundary in memory, and are either 512 bytes in length for filesystem transfers or multiples of that size for swap transfers.

The next lines of code needed to interface a block device driver to the character special environment are **read** and **write** entry points.

```
107:    /*  character special read on id device
108:     */
109:    VOID idread(dev)
110:        DEV dev;
111:        {
112:        IMPORT VOID idstrat();
113:
114:        physio(&idstrat, dev, B_READ);
115:        }
116:
```

```
117:    /*  character special write on id device
118:     */
119:    VOID idwrite(dev)
120:        DEV dev;
121:        {
122:        IMPORT VOID idstrat();
123:
124:        physio(&idstrat, dev, 0);
125:        }
```

The **physio** routine in the resident is the workhorse here. It places **dev** into a BUF structure, which it allocates within the system, and marks the BUF as a character special request. It also does the work of enforcing any memory boundaries before calling the strategy routine.

The last routine needed for a character special interface is an **stty/gtty** entry point. Under most implementations this is just an unused I/O call. It is used mostly for setting terminal I/O parameters. Disk-type devices generally ignore this entry point, so it is set to **nodev** in **CDEVSW**.

3.4.7. Polling

It's also possible to interface a device that doesn't use interrupts - the code actually gets simpler. The strategy routine calls the start routine, which doesn't return until the I/O is done. A listing of our ideal disk driver code using polling follows. It is an absolute minimum driver, the kind you would first implement even if you could do better -- just to get IDRIS running sooner. Code that works solidly now, even if a bit low on features, is better than code that doesn't work yet.

```
/* EXAMPLE OF BARE-BONES POLLED DISK DEVICE DRIVER
 */
#include <std.h>
#include <res.h>
#include <bio.h>

/*  things needed by IDRIS
 */
LOCAL TEXT idname[] = {"id0 id1 id2 id3"};
LOCAL TEXT ridname[]={"rid0 rid1 rid2 rid3"};
LOCAL DEVTAB idtab = {0};
BDEVSW idbdev = {&idopen, &idclose, &idstrat, &idtab, idname};
CDEVSW idcdev = {&idopen, &idclose, &idread, &idwrite, &idsgtty};

/*  things needed by driver
 */
#define NDRIVES 4       /* max drives per controller */
#define MAXBLOCKS 20480 /* 10mb drive */
#define DEVADDR 0xFA80  /* address of device controller regs */
#define RDCMD 0x1A
#define WRTCMD 0x1B
#define DRVBUSY (pv->id_godoit & WORKING)
#define WORKING 1

/*  device happens to have two error registers
 *  and a separate register to start controller
 */
typedef struct {
    BITS id_cmd;    /* add hex 100 for interrupt enable */
    BITS id_error;
    BITS id_cmderr;
    BITS id_drvsel; /* 1 bit per drive (low 4 bits) */
    BITS id_online; /* 1 bit per drive (0 iff online)*/
    BITS id_godoit; /* turn on low-order bit, reset when op done */
    UCOUNT id_memadd;   /* low 16 bits of memory address */
    UCOUNT id_xtradd;   /* which 64KB page */
    BYTES id_bytcnt;    /* neg of bytes to xfer (counts up to 0 ) */
    BYTES id_secsiz;    /* variable sector size not needed */
    BYTES id_secnum;    /* which sector (also positions heads) */
    BYTES id_secoff;    /* offset within sector */
    } IDVEC;
LOCAL IDVEC *contaddr = {DEVADDR};
```

```
/*  close an id-type device
 */
LOCAL VOID idclose(dev)
    DEV dev;
    {
    if (NDRIVES <= dminor(dev))
        uerror ?????
    }

/*  open an id-type device
 */
LOCAL VOID idopen(dev)
    DEV dev;
    {
    if (NDRIVES < dminor(dev))
        uerror(ENXIO)
    }

/*  character special read on id device
 */
LOCAL VOID idread(dev)
    DEV dev;
    {
    IMPORT VOID idstrat();

    physio(&idstrat, dev, B_READ);
    }

/*  disallow sgtty calls on id devices
 */
LOCAL VOID idsgtty(dev, getfl)
    DEV dev;
    BOOL getfl;
    {
    nodev();
    }
```

```
/*  start I/O on id drive
 */
LOCAL VOID istrat(pb)
    FAST BUF *pb;
    IMPORT IDVEC *contaddr;
    FAST IDVEC *pv = *contaddr;

        pv->id_drvsel = dminor(pb->b_dev);
        pv->id_secsiz = 256;
        pv->id_secnum = (pb->b_blkno << 1) | (pb->b_resid >> 8);
        pv->id_secoff = pb->b_resid & 0xff;
        pv->id_memadd = pb->b_phys & 0xffff;
        pv->id_xtradd = pb->b_phys >> 16;
        pv->id_bytcnt =
            -pb->b_count;       /* incrementing counter */
        pv->id_cmd = (pb->b_flag & B_READ) ? RDCMD : WRTCMD;
        pv->id_godoit = YES;
        while (DRVBUSY)
            ;                   /* HOT LOOP */
        if (pv->id_cmderr)      /* no fancy retries */
            {
            ioerror(pb, pv->id_error, pv->id_cmd)
            pb->b_flag |= B_ERROR;
            }
        pb->b_resid = -pv->id_bytcnt;
        idtab.d_nerr = 0;
        iodone(pb);
}
```

```
/*  character special write on id device
 */
LOCAL VOID idwrite(dev)
    DEV dev;
    {
    IMPORT VOID idstrat();

    physio(&idstrat, dev, 0);
    }

/*  end of sample disk driver */
```

3.5. Terminal Driver Tutorial

This tutorial covers the common aspects of terminal driver construction.

3.5.1. Establishing Terminal Driver Entry Points

The following synopsis lists each of the entry points into the device driver that IDRIS will use to do terminal I/O.

```
typdef struct {
    VOID (*d_open)();
    VOID (*d_close)();
    VOID (*d_read)();
    VOID (*d_write)();
    VOID (*d_sgtty)();
    TEXT *d_cname;
    } CDEVSW;
```

The driver entry points are made known via a single address which is plugged into the **chrdevs** table in **main.c**. The code to set up the driver is:

```
/*  IDEAL TERMINAL DRIVER
 *  Copyright (c) 1984 by Ideal Terminal Corporation
 *  a wistful subsidiary of Whitesmiths, Ltd.
 */
#include <std.h>
#include <res.h>
#include <bio.h>
#include <cio.h>

CDEVSW itcdev = {
   &itopen, &itclose, &itread, &itwrite, &itsgtty, itnm};

LOCAL TEXT itnm[] = {"it0 it1 it2"};
```

The interrupt routine addresses are reference is **a.s**, the interrupt module.

3.5.2. Opening a Terminal

The first routine to write in constructing a terminal driver is the open routine. open is called each time a device driver is to be opened. Often it is done via the /odd/log utility process, once for each terminal set up in the file **/adm/init**. The /bin/write communication utility, as well as other user programs, will also open terminals. Thus it is possible to have the same terminal opened multiple times.

For a first try the code to open a terminal might look like:

```
100:  /*    open an id type terminal
101:   */
102:  LOCAL VOID itopen(dev)
103:      DEV dev;
104:      {
105:      IMPORT BITS ttyps;
106:      IMPORT ITVEC *itaddr[];
107:      IMPORT TTY ittty[];
108:      IMPORT VIOD itrint(), itstart();
109:      ITVEC *pv;
110:      TTY *pv;
111:
112:      if (NTERMS <= dminor(dev))
113:          {
114:          uerror(ENXIO);
115:          return;
116:          }
117:      pv = itaddr[dminor(dev)];
118:      pt = &ittty[dminor(dev)];
119:      settyp(dev, YES);
120:      spl(ttyps);
121:      if (!(pt->t_stat & T_OPEN))
122:          {
123:          pt->t_start |= T_OPEN|T_CARR;
124:          if(!pt->t_go)
125:              {
126:              pt->t_go = &itstart;
127:              pt->t_dev = dev;
128:              pt->t_erase = CERASE;
129:              pt->t_kill = CKILL;
130:              pt->t_flag = M_XTABS|M_ECHO|M_CRTLF;
131:              pt->t_speeds = 0x0909; /* 1200 baud */
132:              pv->uart_baud = BAUD1200;
133:              }
134:          pv->uart_cmd = RECV_INT|XMIT_INT|RAISE_DTR;
135:          }
136:      ++pt->t_open;
137:      spl0();
138:      }
```

The routine imports several variables of interest: **itaddr** is an array initialized to the addresses of each controller. **ittty** is an array of the same size as **itaddr**. It contains an instance of a TTY control structure, one for each terminal.

Besides calling **settyp** to set up a possible controlling terminal for the invoking process, the open routine does some other important work. If the terminal is not already open, the routine sets up the initial values for erase, kill, and baud rates.

3.5.3. Terminal Close Routine

The next routine to implement is the close routine. It is called once for each close request on a file descriptor associated with it previously via open. Program termination also causes a close.

Here is the code:

```
139:  /*   close an it type terminal
140:   */
141:  LOCAL VOID itclose(dev)
142:       DEV dev;
143:       {
144:       IMPORT ITVEC *itaddr[];
145:       IMPORT TTY ittty[];
146:       FAST TTY *pt = &ittty[dminor(dev)];
147:       ITVEC *pv = &itaddr[dminor(dev)];
148:
149:       if (--pt->t_open)
150:            return;
151:       wflush(pt);
152:       pt->t_stat = 0;
153:       settyp(dev, NO);
154:       }
155:
```

On the last close of a particular terminal, **itclose** flushes any buffers, clears the status, and releases the controlling terminal. Only the super-user can get rid of a controlling terminal, however.

3.5.4. Read and Write for a Terminal

By far the easiest routines are the **read** and **write** entry points for a terminal driver. The minor device code has already been validated (by **open**) and catch-all routines are ready to go.

Here is the code:

```
159:  /*    read an ideal terminal
160:   */
161:  LOCAL VOID itread(dev)
162:       DEV dev;
163:       {
164:       IMPORT TTY ittty[];
165:
166:       ttread(&ittty[dminor(dev)]);
167:       }
168:
169:  /*    write an ideal terminal
```

```
170:   */
171:  LOCAL VOID itwrite(dev)
172:      DEV dev;
173:      {
174:      IMPORT TTY ittty[];
175:
176:      ttwrite(&ittty[dminor(dev)]);
177:      }
```

3.5.5. STTY/GTTY Call

The next entry point is called as a result of an **stty** or **gtty** system call to set or get terminal parameters. **stty** is also used to clear any type-ahead, to turn off echoing for passwords and such, and to set terminal speed.

```
178:  /*     set or get tty mode
179:   */
180:  LOCAL VOID itsgtty(dev, getfl)
181:      DEV dev;
182:      BOOL getfl;
183:      {
184:      IMPORT ITVEC *itaddr[];
185:      IMPORT BITS baudmap[];
186:      IMPORT TTY ittty[];
187:      FAST BITS speed;
188:      FAST ITVEC *pv = itaddr[dminor(dev)];
189:      FAST TTY *pt = &ittty[dminor(dev)];
190:
191:      if (ttset(pt, getfl))
192:         return;
193:      if (speed = pt->t_speeds & S_ISPEED)
194:          pv->uart_baud = baudmap[speed];
195:      else
196:          pv->uart_cmd = HANGUP;
197:      }
```

Again, an internal call into IDRIS does most of the work. **ttset** does everything but set baud rates. If **getfl** was YES, we don't want to change anything at all, just copy back the old values.

baudmap is presumably an array initialized to values that can be used to tell the **uart** what speed is desired.

3.5.6. The Terminal Driver Name

The last entry point that IDRIS needs is a list of names for the terminals. The name string is a space-separated list of the names which should appear in the /dev directory. Spaces are significant, as extra spaces skip over unused unit numbers.

```
198:  TEXT nmit[] = {"it0 it1 it2"};
199:  ITVEC *itaddr[] = {
200:      0xfff0,
201:      0xffe0,
202:      0xffd0};

203:  #define NTERMS (sizeof (itaddr) / sizeof (itaddr[0]))
204:  TTY ittty[NTERMS] = {0};
205:
206:  BITS baudmap[] = {
207:  /*  50,  50,  75, 110, 134.5, 150,  50, 300 */
208:     0x0, 0x0, 0x1, 0x2,   0x3, 0x4, 0x0, 0x5,
209:  /*  600, 1.2k, 1.8k, 2.4k, 4.8k, 9.6k, 7.2k, 19.2k */
210:    0x6,  0x7,  0x8,  0xa,  0xc,  0xe,  0xd,   0xf };
```

The definition for **baudmap** will remap most of the speeds selected through an **stty** into a close equivalent on the **uart**. The **define** automatically determines the NTERMS parameter used before in the **itopen** routine.

At this point there are at least two unanswered questions: How do the characters get into IDRIS so **ttread** can pick them up? How are they sent out from a **ttwrite**? These questions are dealt with in the discussion below on receiver interrupts.

3.5.7. Receiver Interrupts

When a character is received, the **uart** will send an interrupt to be handled by our receiver interrupt handler. This will eventually wind up calling **ttin** to save up or otherwise process the character for the next ttread.

The code to do this follows:

```
211:  /*    receiver interrupt
212:   */
213:  VOID itrint(dev)
214:      DEV dev;
215:      {
216:      IMPORT ITVEC *itaddr[];
217:      IMPORT TTY *ittty[];
218:      FAST ITVEC *pv = itaddr[dminor(dev)];
219:      FAST TTY *pt = &ittty[dminor(dev)];
220:      FAST BITS status = pv->uart_stat;
221:
222:      if (status & (FRAME_ERR|OVERRUN_ERR))
223:         {
224:          if (status & (FRAME_ERR)
225:              pt->t_speeds |= S_IBREAK;
226:          if (status & OVERRUN_ERR)
227:              pt->t_speeds |= S_ILOST;
228:          ttin(CINTR, pt);
229:          }
230:      if (status & INPUT_READY)
231:          ttin(pv->uart_data, pt);
```

```
232:      }
```

3.5.8. Transmitter Interrupts

ttwrite queues up all characters to be written to a terminal. The characters are dequeued when **ttwrite** calls the routine named in the TTY member **t_go**; here it is **itstart**. When the hardware is finished with a character, **itxint** will be called to signal the completion. **itstart** will also be called from a **ttread** if echoing is turned on (full duplex) and **ttin** is called from the **itrint** routine above.

The **itstart** code follows:

```
233:  /* start output transmitter
234:   */
235:  LOCAL VOID itstart(pt)
236:      FAST TTY *pt;
237:      {
238:      IMPORT ITVEC *itaddr[];
239:      IMPORT TEXT cmaptab[];
240:      IMPORT UCOUNT out_lo;
241:      IMPORT VOID ttrstart();
242:      FAST COUNT c;
243:      FAST ITVEC *pv = itaddr[dminor(pt->t_dev)];
244:
245:      if (!(pv->uart_stat & XMIT_READY))
246:              return;
247:      if (pt->t_stat & (T_STOP|T_TIMER) ||
248:          (c = deqc(&pt->t_outq)) < 0)
249:          {
250:          pv->uart_cmd &= ~XMIT_INT;
251:          return;
252:          }
253:      if (pt->t_flag & M_RAW)
254:          pv->uart_data = c;
255:      else if (c < 0200)
256:          {
257:          switch (pt->flag & (M_EVEN|M_ODD)
258:             {
259:          case M_EVEN|M_ODD:
260:             c |= 0200;
261:             break;
262:          case M_EVEN:
263:             c |= cmaptab[c] & 0200;
264:             break;
265:          case M_ODD:
266:             c |= (cmaptab[c] & 0200) ^ 0200;
267:             }
              pv->uart_cmd |= XMIT_INT
268:          pv->uart_data = c;
269:          }
270:      else
```

```
271:            {
272:            timeout(&ttrstart, pt, c & 0117);
273:            pt->t_stat |= T_TIMER;
274:            pv->uart_cmd &= ~XMIT_INT;
275:            }
276:        if (pt->t_outq.c_num == 0 || pt->t_outq.c_num == out_lo)
277:            wakeup(&pt->t_outq);
278:        }
```

There is a quick check at line 245 to see if the transmitter is ready to accept a character. This function gets called for every character queued on write system calls and receiver interrupts, **ttin**.

T_STOP indicates if X_OFF was received and T_TIMER indicates whether formfeed, carriage return or other delays are in progress.

The command to turn off transmit interrupts at line 250 and 254 is not always necessary. Some **uarts** generate a single interrupt when their transmit buffers go empty; other **uarts** keep the transmit interrupt request jammed on until a character is supplied. This code handles the latter. The transmit interrupt enable must be reset if there is no character to send. You can observe the behavior of IDRIS to debug your transmit interrupt routine:

- if you get 1, 2 or 3 characters of output, and hitting keys on the keyboard causes a few more to come out, then you have no transmitter interrupts;

- if you see all output and echoing of the keyboard, but nothing will run (no disk activity) then the transmitter interrupt is running constantly. Sometimes the manufacturer's data sheet will shed some light about their chip's interrupt behavior; the wise system programmer will observe the system's behavior nonetheless.

8-bits are output in raw mode, lines 253 and 254; and 7-bits with parity are output in cooked mode, lines 255 to 269. Delay codes are handled in lines 270 to 279.

Lines 276 and 277 cause any program roadblocked on output to the terminal to resume execution.

When the transmission is complete, **itstart** must to be called. Since the uart can send an interrupt on buffer empty, we will handle it in **itxintr**.

The code is:

```
279:  /*    handle transmitter empty interrupt
280:   */
281:  VOID itxintr(dev)
282:      DEV dev;
283:      {
284:      IMPORT TTY ittty[];
285:      FAST TTY *pt = &ittty[dminor(dev)];
286:
287:      itstart(pt);
288:      }
```

This will "kick" the start routine to send the next character. If the receiver and transmitter are assigned the same interrupt, move line 287 to the point directly after line 231 above.