12 An Example Device Driver

Overview

- Device driver: a set of procedures **controlling a peripheral hardware device**.
- Device driver routines are partitioned into:
  - **Upper-half** device drivers: called from **user programs**.
  - **Lower-half** device drivers: handling device **interrupts**.

  These two communicate via a shared data structure, the **device control block**.
- In this chapter we discuss a device driver for a **teletype terminal (tty)**, a standard computer terminal with a keyboard managing:
  - output to display screen,
  - input from keyboard,

  using a SLU, an asynchronous serial line unit controller (see §2.1.9 and §2.1.11)

Overview

- Upper-half tty device routines implement the `read`, `write`, `getc`, `putc` and `control` **device-independent I/O procedures**.
- Lower-half tty device routines consist of:
  - **Output procedure `ttoin`**: called when transmitter interrupts, sending a character from the out-buffer – i.e., from the queue of characters waiting to be sent.
  - **Input procedure `ttyiin`**: called whenever an input character arrives, puts it in a queue (in-buffer) of incoming characters to be retrieved by upper-half routines.
  - **Initialization procedure `ttyinit`**: fills in the so-called **device control block** and interrupt vectors when system starts.

Introduction

- The **device switch table** **`devtab`** forms the framework linking interrupts, devices and device driver routines.
- This chapter introduces the device driver routines for a standard computer terminal with a keyboard, called **teletype (tty)**.
- From the computer's viewpoint, this device consists of a **asynchronous serial line unit (SLU)**, between the keyboard and the system bus. I.e. a computer terminal connected to the computer over an asynchronous serial line unit (SLU) – see §2.1.8 to §2.1.11

12.1 The Device Type TTY

- Each tty has as input device a keyboard **transmitting characters** (which are typed) to the computer, and an output device for displaying characters received from the computer on the screen.
- The tty device driver consists of routines which **map operating system routines** for reading/writing characters to **hardware operations** transmitting characters to the display and receiving characters from keyboard.
- In practice, these driver routines communicate with the SLU-controller, requesting it to **send characters** or to **receive chars**.
- The controller's hardware transforms characters to and from **electrical impulses** on lines connecting the SLU to the terminal.

Name	**High-Order Byte**	**Low-Order Bytes**
`RCSR`	unused	receiver control
`RBUF`	receiver error	received data
`XCSR`	unused	transmitter control
`XBUF`	unused	transmitted data buffer

The Device Type TTY

- To minimize the **interference** between I/O devices and running processes, the driver uses **interrupt-driven processing**, to:
  1. transmit characters when the SLU is **idle**,
  2. read characters when these are **received** by the SLU,
  3. handle **errors** in the receiving process (receiver), and
  4. coordinate requests for I/O with the **speed** of the device.
- The latter is important because character transmission times are **many orders of magnitude slower** than processing speed of the CPU!

The Device Type TTY

The tty driver operates using **parameters**, so it can be used for a variety of terminals in a variety of system configurations. E.g.,

- Several parameters control the **echo** of characters typed on the keyboard onto the screen.
  - Terminals operating in **full-duplex mode** do not display characters typed by the user – such terminals need the system to echo these characters.
  - Terminals operating in **half-duplex mode** display those characters automatically, and don't need the system to echo them.
- There are parameters concerning whether to echo **unprintable control chars** as printable combinations.
- Some parameters are dealing with **moving to a new line**: There are two unprintable characters controlling movement of the cursor:
  - **return** (also called **carriage return**): moving the cursor to the beginning of the current line
  - **newline** (`"\n"`): moving the cursor vertically down by one line

The Device Type TTY

- However, programs like to deal with a **single end-of-line character** to move to a new line. Hence, in practice, terminal must receive **both** **`newline`** and **`return`** characters to move to the start of a new line.
- Now, to simplify programming, the tty driver can interpret both **`newline`** and **`return`** as the *move-to-the-start-of-a-new-line* character, depending on the value of several **parameters**:
  - **`icrlf`**: controls mapping of **`return`** to *move-to-the-start-of-a-new-line*, when received from the keyboard.
  - **`ocrlf`**: controls mapping of **`newline`** to *move-to-the-start-of-a-new-line*, i.e., controls whether the driver inserts **`return`** whenever a program writes **`newline`**.

12.2 Upper and Lower Halves of the Device Driver

- The tty device drivers are partitioned in two sets:
  - **Upper half** device drivers: These are called by user processes through `devtab` in order to read/write characters.
  - **Lower half** device drivers: These are called by the interrupt dispatcher through the `intmap` interrupt dispatch table during interrupts, in order to transfer characters - these do the real work.

  This partition decouples **normal processing** from **hardware interrupts**.

Upper and Lower Halves of the Device Driver

- Every device has its own **circular buffers**:
  - One for **transferring input requests**, connecting high-level calls for actions for input along the SLU, and
  - one for **transferring high-level output requests** to actions for output along the SLU.
- I.e., these buffers record **outgoing data** from the time the user requests them to be sent until the time the device (SLU) receives them; and also record **incoming data** from the time the SLU deposits them until the user program requests them.

Upper and Lower Halves of the Device Driver

These buffers are important for three reasons:

1. The driver can accept incoming characters **before** the user process reads them (since the terminal user can **start typing at any time**).
2. Device such as disks transfer data in **large blocks**.
3. These buffers permit the drivers to perform I/O **concurrently** with user processes. I.e., when a user process writes data, the driver copies them into a buffer, allowing the user to **continue executing** while the driver transfers the data from the buffer to the device.

Device Driver's Upper and Lower Halves

So the tty driver uses **two circular buffers per terminal**: One for the **output**:

- Output operations issued by the calling (user) program deposit to-be-written in the output buffer, and return to the caller.
- Also, these operations are accompanied by setting the interrupt-enable-bit of the SLU to 1 for the output line.
- As a result, the SLU transmitter interrupts whenever it is idle, calling interrupt processing routines in the lower half of the tty device driver.
- It picks up characters from the output buffer and places them into the output control area, thereby starting their transmission.

And one for the **input**:

- Whenever the SLU receiver interrupts after/when it has received a character, the interrupt dispatcher calls the lower-half input interrupt routine.
- The lower-half interrupt input handler reads the waiting character and deposits it in the circular buffer.
- A process waiting for input from the (empty) circular buffer is started as soon as the next character arrives.

Upper and Lower Halves of the Device Driver

- **Ideally**, the two halves of the device driver communicate through the shared buffer:
  - Upper-half routines **enqueue requests** for data transfer or device control; they do not interact with devices directly.
  - Lower-half routines **transfer data from buffer** or control devices; they do not interact with user program directly.
- **In practice**, the two halves do more than manipulating shared data:
  - An output routine **starts up** the lower-half routine when it deposits output in its buffer.
  - In case the available buffer space is **full** when a process tries to write, the upper and lower halves must **coordinate**, stopping the process until space becomes available.
  - Also these routines start a process waiting for output from an empty buffer, as soon as the **next character arrives**.

12.3 Synchronization of the Upper and Lower Halves

- The synchronization of the upper and lower halves via circular buffers seems at first glance to be a simple instance of **producer-consumer synchronization**, implemented using semaphores; for upper-half output routines **produce** characters **consumed** by the lower-half output routines; and lower-half input routines **produce** characters **consumed** by upper-half input routines.
- Input poses no problem because user processes calling upper-half routines can **wait** for lower half input routines to produce a character. The latter is **signalled** by a `signal` operation, in interrupt mode.
- However, the following violates the rule that **processes in interrupt mode** should always be `current` or `ready`, and never be on an any other list than the ready list:

  For output poses a problem, if output offered by a user process is translated in a `signal` operation in the upper-half routines, since this implies that the lower-half routines should `wait` by a `wait` operation, until that a character becomes available! And that implies that, i.e., the **null process** would be put on a semaphore queue, which is **forbidden**.

A Problem ... and its Solution!

Problem:

- During an interrupt, the process carrying out that interrupt, should always be running (`CURRENT`) or on the ready queue (`READY`), but never be waiting on another queue, i.e. be in another process state. But that is not implied by the above producer/consumer synchronization for output operations!
- For the null process is only allowed to be in the `READY` or `CURRENT` state, since, otherwise, there might be no process ready to run when calling **`resched`**!

Solution for the output:

- Instead of synchronizing the upper and lower halves by the **availability** of output characters, occupying free buffer space, synchronize them by the size of the complement: **The number of free buffer slots**!
- Now the output operation in the upper-halve device driver has to **wait** for free space to become available, until the output operation in the lower-half **signals** that free space has become available, after an output operation in the SLU has interrupted, i.e., has ended.
- In this way, the lower-half **produces** free space, which upper-half **consumes**, when implementing an output operation.

12.4 Control Block and Buffer Declarations

- Each SLU used as tty has its own pair of input and output **buffers**, and accompanying input and output **semaphores** `isem` (counting number of characters) and `osem` (counting number of free buffer slots).
- This, and other information, is kept in the tty control block structure. There is a control block structure for each tty device. Control blocks also contain, e.g., the control parameters `icrlf` and `ocrlf`.

Control Block and Buffer Declarations

```
/* tty.h */

#define IOCHERR         0200            /* bit set in when an error     */
                                        /* occurred reading the char    */
#define OBMINSP         20              /* min space in buffer before   */
                                        /* processes awakened to write  */
#define EBUFLEN         20              /* size of echo queue           */

/* size constants */

#ifndef Ntty
#define Ntty            1               /* number of serial tty lines   */
#endif
#ifndef IBUFLEN
#define IBUFLEN         128             /* num. chars in input queue    */
#endif
#ifndef OBUFLEN
#define OBUFLEN         64              /* num. chars in output queue   */
#endif

/* mode constants */

#define IMRAW           'R'             /* raw mode => nothing done     */
#define IMCOOKED        'C'             /* cooked mode => line editing  */
#define IMCBREAK        'K'             /* honor echo, etc, no line edit*/
#define OMRAW           'R'             /* raw mode => normal processing*/
```

Control Block and Buffer Declarations

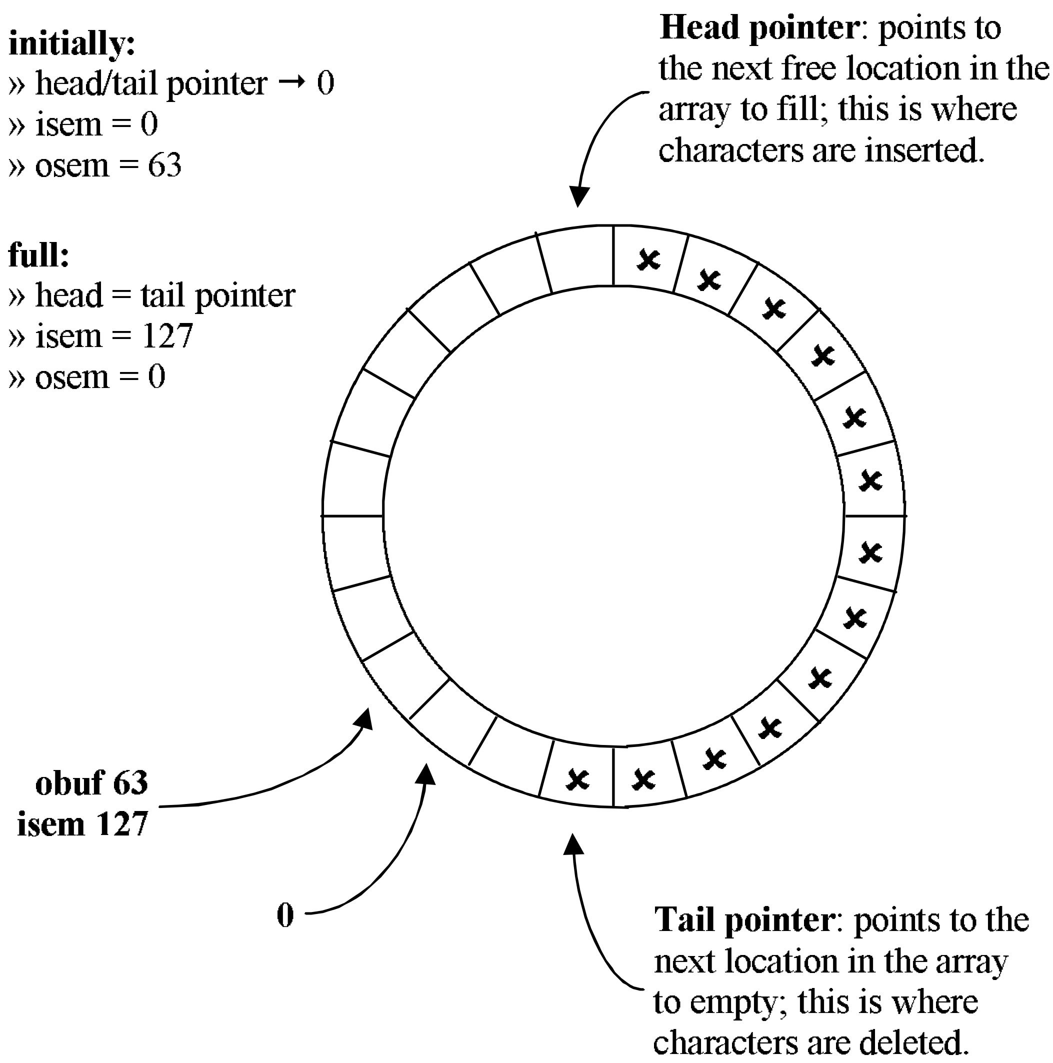
```
struct  tty     {                       /* tty line control block       */
        int     ihead;                  /* head of input queue          */
        int     itail;                  /* tail of input queue          */
        char    ibuff[IBUFLEN];         /* input buffer for this line   */
        int     isem;                   /* input semaphore              */
        int     ohead;                  /* head of output queue         */
        int     otail;                  /* tail of output queue         */
        char    obuff[OBUFLEN];         /* output buffer for this line  */
        int     osem;                   /* output semaphore             */
        int     odsend;                 /* sends delayed for space      */
        int     ehead;                  /* head of echo queue           */
        int     etail;                  /* tail of echo queue           */
        char    ebuff[EBUFLEN];         /* echo queue                   */
        char    imode;                  /* IMRAW, IMCBREAK, IMCOOKED    */
        Bool    iecho;                  /* is input echoed?             */
        Bool    ieback;                 /* do erasing backspace on echo?*/
        Bool    evis;                   /* echo control chars as ^X ?   */
        Bool    ecrlf;                  /* echo CR-LF for newline?      */
        Bool    icrlf;                  /* map '\r' to '\n' on input?   */
        Bool    ierase;                 /* honor erase character?       */
        char    ierasec;                /* erase character (backspace)  */
        Bool    ikill;                  /* honor line kill character?   */
        char    ikillc;                 /* line kill character          */
        int     icursor;                /* current cursor position      */
        Bool    oflow;                  /* honor ostop/ostart?          */
        Bool    oheld;                  /* output currently being held? */
        char    ostop;                  /* character that stops output  */
        char    ostart;                 /* character that starts output */
        Bool    ocrlf;                  /* echo CR/LF for LF ?          */
        char    ifullc;                 /* char to send when input full */
        struct  csr     *ioaddr;        /* device address of this unit  */
};
extern  struct  tty tty[];
```

Control Block and Buffer Declarations

```
#define BACKSP  '\b'                    /* backspace one character pos. */
#define BELL    '\07'                   /* usually an audiable tone     */
#define ATSIGN  '@'
#define BLANK   ' '
#define NEWLINE '\n'                    /* line feed                    */
#define RETURN  '\r'                    /* carriage return              */
#define STOPCH  '\023'                  /* control-S stops output       */
#define STRTCH  '\021'                  /* control-Q restarts output    */
#define UPARROW '^'                     /* usually for visuals like ^X  */

/* ttycontrol function codes */
#define TCSETBRK        1               /* turn on BREAK in transmitter */
#define TCRSTBRK        2               /* turn off BREAK "       "     */
#define TCNEXTC         3               /* look ahead 1 character       */
#define TCMODER         4               /* set input mode to raw        */
#define TCMODEC         5               /* set input mode to cooked     */
#define TCMODEK         6               /* set input mode to cbreak     */
#define TCICHARS        8               /* return number of input chars */
#define TCECHO          9               /* turn on echo                 */
#define TCNOECHO        10              /* turn off echo                */
#define TFULLC          BELL            /* char to echo when buffer full*/
```

Control Block and Buffer Declarations

- **Key components:**
  - input buffer `ibuff` and input semaphore `isem`
  - output buffer `obuff` and output semaphore `osem` (counting the number of free buffer slots) – see the figure below:



Control Block and Buffer Declarations

- There is one **control block structure** per device in array tty, indexed by the minor device number.
- `Ntty` is the **number of tty devices**: the system configuration program assigns a minor device number from 0 ... `ntty` − 1 to each device.
- The system configuration program also places the minor device number in the **device switch table** `devtab`.
- Both the interrupt device routines in the lower half and the driver routines in the upper half use minor device numbers as index into array tty.
- Control block contains the information needed for synchronizing the upper and lower half routines.

12.5 Upper-Half tty Input Routines

- Routines

  `ttygetc` `ttyputc` `ttyread` `ttywrite`

  form basis of upper-half tty driver routines, corresponding to:

  `getc` `putc` `read` `write`

  which are the high-level I/O operations from §11.

- **Simplest** is `ttygetc`:
  - `ttygetc` retrieves the minor device number from device switch table and uses it as index into array tty to locate the correct corresponding control block.
  - Then it executes `wait(isem)` until the lower-half input routine deposits a character in **ibuf**.
  - When `wait` returns, `ttygetc` extracts next character from **ibuf** and updates tail pointer using modulo (128) arithmetic and returns.

Upper-Half tty Input Routines – ttygetc

```
/* ttygetc.c - ttygetc */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttygetc - read one character from a tty device
 *------------------------------------------------------------------------
 */
ttygetc(devptr)
struct  devsw   *devptr;
{
        char    ps;
        int     ch;
        struct  tty   *iptr;

        disable(ps);
        iptr = &tty[devptr->dvminor];
        wait(iptr->isem);               /* wait for a character in buff */
        ch = LOWBYTE & iptr->ibuff[iptr->itail++];
        if (iptr->itail == IBUFLEN)
                iptr->itail = 9;
        restore(ps);
        return(ch);
}
```

Upper-Half tty Input Routines – ttyread

- `ttyread` has **three parameters**:
  1. a device switch table entry to retrieve the minor device number,
  2. the address of the buffer to which read input data should be output, and
  3. the number of characters to read `nread`.
- **Reading all characters** in `ibuf` is encoded by *number of characters to read equals zero.*
- Two cases are treated:
  1. `bread` $\leq$ number of characters in **ibuf**
  2. `nread` $>$ number of characters in **ibuf**

Upper-Half tty Input Routines – ttyread

```
/* ttyread.c - ttyread, readcopy */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttyread - read one or more characters from a tty device
 *------------------------------------------------------------------------
 */
ttyread(devptr, buff, count)
struct  devsw   *devptr;
int count;
char *buff;
{
        char ps;
        register struct tty *iptr;
        int avail, nread;
```

Upper-Half tty Input Routines – ttyread

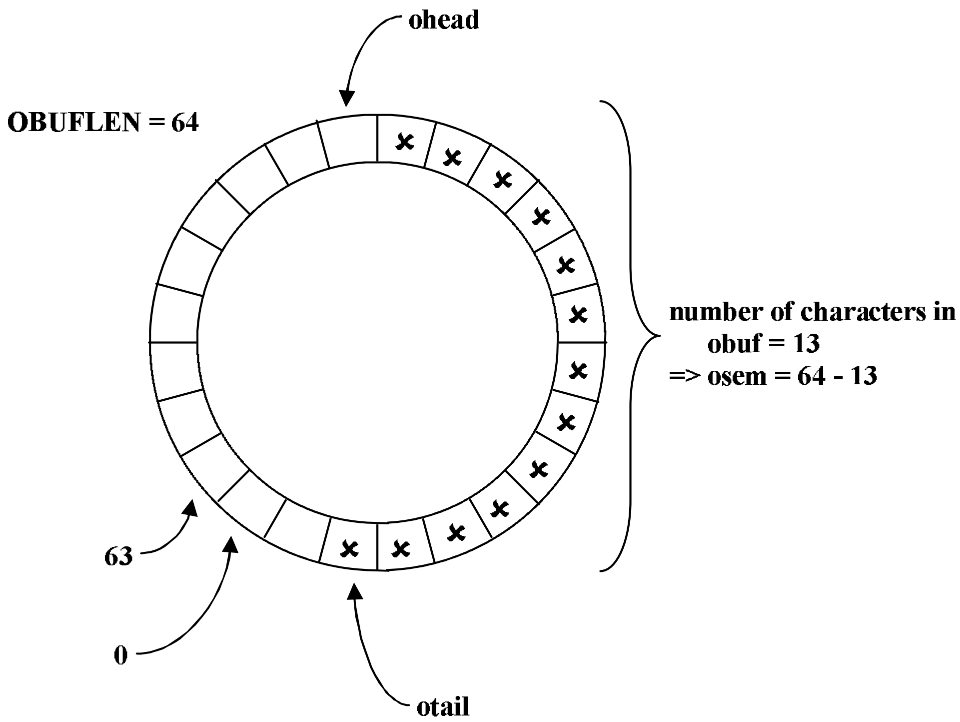
ttyread continued:

```
        if (count < 0)
                return(SYSERR);
        disable(ps);
        avail=scount( (iptr= &tty[devptr->dvminor])->isem );
        if ( (count = (count==0 ? avail : count)) == 0) {
                restore(ps);
                return(0); // ttyread (..., 0) invokes reading
                           // ALL characters available in the buffer!
        }
        nread = count;
        if (count <= avail)
                readcopy(buff, iptr, count);
                   // if there are enough charachters in the buffer
                   // to satisfy the read request, copy them directly
                   // to the user's buffer
        else {
                if (avail > 0) {
                        readcopy(buff, iptr, avail);
                        buff += avail;
                        count -= avail;
                }
                for ( ; count>0 ; count--)
                        *buff++ = ttygetc(devptr);
                   // when the user requests more characters than are
                   // present in the buffer, issue additional calls
                   // of ttygetc
        }
        restore(ps);
        return(nread);
}
```

Upper-Half tty Input Routines – ttyread

```
/*------------------------------------------------------------------------
 *  readcopy - high speed copy procedure used by ttyread
 *------------------------------------------------------------------------
 */
LOCAL readcopy(buff, iptr, count)
register char *buff;
struct tty *iptr;
int count;
{
        register char *qtail, *qend, *uend /* copy loop variables */

        qtail = &iptr->ibuff[iptr->itail];
        qend  = &iptr->ibuff[IBUFLEN];
        uend = buff + count;
        while ( buff < uend ) {
                *buff++ = *qtail++;
                if ( qtail >= qend )
                        qtail = iptr->ibuff;
        }
        iptr->itail = qtail-iptr->ibuf;
        sreset(iptr->isem, scount(iptr->isem)-count);
}
```

12.6 Upper-Half `tty` Output Routines



- `ttyputc`: note `iptr->ocrlf` flag indicates whenever after `ch == NEWLINE` the return-character should be inserted by `ttyputc(devptr, return)`.
- Just before `ttyputc` returns, `ttyputc` enables output - transmitter interrupts in the SLU, to guarantee that lower-half will transfer characters inserted into `obuf` (by setting its control bit to 1; if it is 1 already this causes no harm)
- `ctstat` – see page 170 – identifies the transmitter control and status field `XCSR` of the SLU-register – see page 29. Bit 1 of `XCSR` set to 1 enables it to interrupt when transmitter becomes idle.

Upper-Half tty Output Routines

- Structure `csr` defines the layout of the SLU device registers in the address space.
- To move data into SLU device registers, `ttyputc` needs to know the address assigned to the SLU upon hardware installation. This address is put at system configuration time in the device switch table `devtab`.
- During device initialization this is copied into the `tty` control block field `ioaddr`, by calling `ttyinit` (see page 182) via calling `init(o)`.
- Using the device register address `ioaddr` as pointer, `ttyputc` "writes into the `ctstat` field of the structure" it points to. Since this address is "beyond real memory", it is computed by the hardware and the device output interrupt enable bit is assigned to 1 by assigning $\texttt{SLUENABLE} = 0100_8 = 1\,000\,000_2$.
- This is the only way upper-half routines have to awaken low-half routines to initiate character transfer: By setting bit nr. 6 to 1 whenever current output finishes, SLU will interrupt; calling the interrupt dispatcher, which in its turn calls the lower-half output routine, which finds a character in the buffer and transmits it.

Upper-Half tty Output Routines – `slu.h`

```
/* slu.h */

/* standard serial line unit device constants */

#define SLUENABLE       0100            /* device interrupt enable bit  */
#define SLUREADY        0200            /* device ready bit             */
#define SLUDISABLE      0000            /* device interrupt disable mask*/
#define SLUTBREAK       0001            /* transmitter break-mode bit   */
#define SLUERMASK       0170000         /* mask for error flags on input*/
#define SLUCHMASK       0377            /* mask for input character     */

/* SLU device register layout and correspondence to vendor's names      */

struct  csr     {
        int     crstat;         /* receiver control and status  (RCSR)  */
        int     crbuf;          /* receiver data buffer         (RBUF)  */
        int     ctstat;         /* transmitter control & status (XCSR)  */
        int     ctbuf;          /* transmitter data buffer      (XBUF)  */
};
```

- $\texttt{SLUENABLE} = 0100_8 = 100\,0000_2$
  (device interrupt enable bit)
- $\texttt{SLUREADY} = 0200_8 = 1000\,0000_2$
  (device ready bit)
- $\texttt{SLUERMASK} = 0170000_8 = 1111\,0000\,0000\,0000_2$
  (mask for input errors)
- $\texttt{SLUCHMASK} = 0377_8 = 0000\,0000\,1111\,1111_2$
  (mask for input character)

Upper-Half tty Output Routines – ttyputc

```
/* ttyputc.c - ttyputc */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttyputc - write one character to a tty device
 *------------------------------------------------------------------------
 */
ttyputc(devptr, ch )
struct  devsw   *devptr;
char    ch;
{
        struct  tty   *iptr;
        char    ps;

        iptr = &tty[devptr->dvminor];
        if ( ch==NEWLINE && iptr->ocrlf )
                ttyputc(devptr,RETURN);
        disable(ps);
        wait(iptr->osem);               /* wait for space in queue      */
        iptr->obuff[iptr->ohead++] = ch;
        if (iptr->ohead >= OBUFLEN)
                iptr->ohead = 0;
        (iptr->ioaddr)->ctstat = SLUENABLE;
        restore(ps);
        return(OK);
}
```

Upper-Half tty Output Routines

```
/* ttywrite.c - ttywrite, writcopy */

#include <conf.h> <kernel.h> <tty.h> <io.h> <slu.h>

/*------------------------------------------------------------------------
 *  ttywrite - write one or more characters to a tty device
 *------------------------------------------------------------------------
 */
ttywrite(devptr, buff, count)
struct  devsw   *devptr;
char    *buff;
int     count;
{
        register struct tty *ttyp;
        int ncopied;
        char ps;

        if (count < 0)
                return(SYSERR);
        disable(ps);
        ttyp = &tty[devptr->dvminor];
        if ((avail=scount(ttyp->osem)) >= count) {
                writcopy(buff, ttyp, count);
                (ttyp->ioaddr)->ctstat = SLUENABLE;
        } else {
                if (avail > 0) {
                        writecopy(buff, ttyp, avail);
                        buff += avail;
                        count -= avail;
                }
                for (; count>0 ; count--)
                        ttyputc(devptr, *buff++);
        }
        restore(ps);
        return(OK);
}
```

Upper-Half tty Output Routines – ttyputc

```
/*------------------------------------------------------------------------
 *  writcopy - high-speed copy from user's buffer into system buffer
 *------------------------------------------------------------------------
 */
LOCAL writcopy(buff, ttyp, count)
char    *buff;
struct  tty *ttyp;
int     count;
{
        register char   *qhead, *qend, *uend;

        qhead = &ttyp->obuff[ttyp->ohead];
        qend  = &ttyp->obuff[OBUFLEN];
        uend  = buff + count;
        while (buff < uend) {
                *qhead++ = *buff++;
                if ( qhead >= qend )
                        qhead = ttyp->obuff;
        }
        }
        ttyp->ohead = qhead - ttyp->obuff;
        sreset(ttyp->osem, scount(ttyp->osem)-count);
}
```

12.7 Lower-Half tty Driver Routines

- There are two of them:
  - lower-half input-interrupt routine `ttyiin` (code on later slide): Recall that the interrupt dispatcher calls `ttyoin` with interrupts disabled wherever output device driver is idle (because only the interrupt is generated).
  - lower-half output-interrupt routine `ttyoin`: Recall that `iptr` points to address of tty-control block for the interrupting device.
- A `ttyoin(iptr)`-call considers three cases:
  - a character from echo buffer `ebuf` must be transmitted
  - the driver should do something because output is currently interrupted, as indicated by `oheld`, until line is full
  - a character from the output buffer is transmitted – two ways to do this:
    - ∗ There is more than `OBMINSP` (= 20) free space in `obuf`. Character is put in device register and `signal(osem)` is called to free **one buffer slot**.
    - ∗ There is less than `OBMINSP` free space in `obuf`. Chars are still transmitted, but `signal(osem)` is delayed to halt (potentially) calls of `ttyputc`.

Lower-Half tty Driver Routines

```
/* ttyoin.c - ttyoin */

#include <conf.h> <kernel.h> <tty.h> <io.h> <slu.h>

/*------------------------------------------------------------------------
 *  ttyoin  --  lower-half tty device driver for output interrupts
 *------------------------------------------------------------------------
 */
INTPROC ttyoin(iptr)
        register struct tty   *iptr;
{
        register struct csr     *cptr;
        int     ct;

        cptr = iptr->ioaddr;
        if (iptr->ehead != iptr->etail) {
                cptr->ctbuf = iptr->ebuff[iptr->etail++];
                if (iptr->etail >= EBUFLEN)
                        iptr->etail = 0;
                return;
        }
        if (iptr->oheld) {                      /* honor flow control   */
                cptr->ctstat = SLUDISABLE;
                return;
        }
        if ((ct=scount(iptr->osem)) < OBUFLEN) {
                cptr->ctbuf = iptr->obuff[iptr->otail++];
                if (iptr->otail >= OBUFLEN)
                        iptr->otail = 0;
                if (ct > OBMINSP)
                        signal(iptr->osem);
                else if ( ++(iptr->odsend) == OBMINSP) {
                        iptr->odsend = 0;
                        signaln(iptr->osem, OBMINSP);
                }
        } else
                cptr->ctstat = SLUDISABLE;
}
```

12.7.2 Lower-Half Input Processing

- Input interrupt processing is **more complex** because it caters for **character echo, line editing, processing of input errors** and **buffer overflow**.
- It operates in one of three modes, specified by `imode` field in tty control block:
  - `raw`: accumulates characters in `ibuff` without further processing.
  - `cooked`: does **character echo**; honors suspend or restart output; accumulates **full lines** before passing them on to upper-half routines; honors **input editing** by erasing previous characters or killing entire lines.
  - `cbreak`: honors **control chars** but does **no line editing**.

1. In `raw` mode:
   - `ttyiin` takes **input character** from SLU-receiver unit
   - deposits it in `ibuf`,
   - signals `isem`,
   - if no space remains, `ttyiin` throws character away; error is reported by setting 8^{th} character bit to 1.

Lower-Half Input Processing

2. In `cbreak` mode,
   - `ttyiin` takes input character from SLU-receiver unit,
   - maps return to newline,
   - handles output flow control:
     - field `oflow` determines whether driver honours flow control,
     - if it does, driver **suspends** output by setting `oheld` when receiving `ostop` character,
     - starting output when receiving `ostart` character,
     - `ostart` and `ostop` are control characters, not placed in `ibuf` for processing by upper-half input routine `ttygetc`,
     - reports `ibuffer` overflow by sending `ifullc` character in echo buffer, **ringing a bell**, and does character echo.

Lower-Half Input Processing

3. In `cooked` mode, line editing is also performed:
   - accumulating lines in `ibuf` using var `icursor` for counting chars on current line
   - erasing chars when `ierasec` arrives (decrementing `icursor` by 1)
   - when receiving line-kill character `ikillc`, `ttyiin` backs over all chars on current line by `icursor := 0`,
   - calling `erase1` procedure to obliterate chars from display
   - upon receiving `NEWLINE` or `RETURN` char, signalling `isem` `icursor` times

Lower-Half Input Processing

```
/* ttyiin.c ttyiin, erase1, eputc, echoch */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttyiin  --  lower-half tty device driver for input interrupts
 *------------------------------------------------------------------------
 */
INTPROC ttyiin(iptr)
        register struct tty     *iptr;  /* pointer to tty block         */
{
        register struct csr *cptr;
        register int    ch;
        Bool    cerr;
        int     ct;

        cptr = iptr->ioaddr;
        if (iptr->imode == IMRAW) {
                if (scount(iptr->isem) >= IBUFLEN) {
                        ch = cptr->crbuf;
                        return;
                }
                if ((ch=cptr->crbuf)&SLUERMASK) /* character error        */
                        iptr->ibuff[iptr->ihead++] = (ch&SLUCHMASK)|IOCHERR;
                else                            /* normal read complete   */
                        iptr->ibuff[iptr->ihead++] = ch & SLUCHMASK;
                if (iptr->ihead >= IBUFLEN)
                        iptr->ihead = 0;
                signal(iptr->isem);
```

Lower-Half Input Processing

```
} else {                                /* cbreak | cooked mode */
        cerr = ((ch=cptr->crbuf)&SLUERMASK) ? IOCHERR : 0;
        ch &= SLUCHMASK;
        if ( ch == RETURN && iptr->icrlf )
                ch = NEWLINE;
        if (iptr->oflow) {
                if (ch == iptr->ostart) {
                        iptr->oheld = FALSE;
                        cptr->ctstat = SLUENABLE;
                        return;
                }
                if (ch == iptr->ostop) {
                        iptr->oheld = TRUE;
                        return; }
        }
        iptr->oheld = FALSE;
        if (iptr->imode == IMCBREAK) {          /* cbreak mode  */
                if (scount(iptr->isem) >= IBUFLEN) {
                        eputc(iptr->ifullc,iptr,cptr);
                        return;
                }
                iptr->ibuff[iptr->ihead++] = ch | cerr;
                if (iptr->ihead >= IBUFLEN)
                        iptr->ihead = 0;
                if (iptr->iecho)
                        echoch(ch,iptr,cptr);
                if (scount(iptr->isem) < IBUFLEN)
                        signal(iptr->isem);
        } else {                                /* cooked mode  */
                if (ch == iptr->ikillc && iptr->ikill) {
                        iptr->ihead -= iptr->icursor;
                        if (iptr->ihead < 0)
                                iptr->ihead += IBUFLEN;
                        iptr->icursor = 0;
                        eputc(RETURN,iptr,cptr);
                        eputc(NEWLINE,iptr,cptr);
                        return;
                }
```

Lower-Half Input Processing

```
                        if (ch == iptr->ierasec && iptr->ierase) {
                                if (iptr->icursor > 0) {
                                        iptr->icursor--;
                                        erase1(iptr,cptr);
                                }
                                return;
                        }
                        if (ch == NEWLINE || ch == RETURN) {
                                if (iptr->iecho)
                                        echoch(ch,iptr,cptr);
                                iptr->ibuff[iptr->ihead++] = ch | cerr;
                                if (iptr->ihead >= IBUFLEN)
                                        iptr->ihead = 0;
                                ct = iptr->icursor+1; /* +1 for \n or \r*/
                                iptr->icursor = 0;
                                signaln(iptr->isem,ct);
                                return;
                        }
                        ct = scount(iptr->isem);
                        ct = ct < 0 ? 0 : ct;
                        if ((ct + iptr->icursor) >= IBUFLEN-1) {
                                eputc(iptr->ifullc,iptr,cptr);
                                return;
                        }
                        if (iptr->iecho)
                                echoch(ch,iptr,cptr);
                        iptr->icursor++;
                        iptr->ibuff[iptr->ihead++] = ch | cerr;
                        if (iptr->ihead >= IBUFLEN)
                                iptr->ihead = 0;
                }
        }
}
```

Lower-Half Input Processing

```
/*------------------------------------------------------------------------
 *  erase1  --  erase one character honoring erasing backspace
 *------------------------------------------------------------------------
 */
LOCAL erase1(iptr,cptr)
        struct  tty   *iptr;
        struct  csr    *cptr;
{
        char    ch;

        if (--(iptr->ihead) < 0)
                iptr->ihead += IBUFLEN;
        ch = iptr->ibuff[iptr->ihead];
        if (iptr->iecho) {
                if (ch < BLANK || ch == 0177) {
                        if (iptr->evis) {
                                eputc(BACKSP,iptr,cptr);
                                if (iptr->ieback) {
                                        eputc(BLANK,iptr,cptr);
                                        eputc(BACKSP,iptr,cptr);
                                }
                        }
                        eputc(BACKSP,iptr,cptr);
                        if (iptr->ieback) {
                                eputc(BLANK,iptr,cptr);
                                eputc(BACKSP,iptr,cptr);
                        }
                } else {
                        eputc(BACKSP,iptr,cptr);
                        if (iptr->ieback) {
                                eputc(BLANK,iptr,cptr);
                                eputc(BACKSP,iptr,cptr);
                        }
                }
        } else
                cptr->ctstat = SLUENABLE;
}
```

Lower-Half Input Processing

```
/*------------------------------------------------------------------------
 *  echoch  --  echo a character with visual and ocrlf options
 *------------------------------------------------------------------------
 */
LOCAL echoch(ch, iptr, cptr)
        char    ch;             /* character to echo                    */
        struct  tty   *iptr;    /* pointer to I/O block for this devptr */
        struct  csr    *cptr;   /* csr address for this devptr          */
{
        if ((ch==NEWLINE||ch==RETURN)&&iptr->ecrlf) {
                eputc(RETURN,iptr,cptr);
                eputc(NEWLINE,iptr,cptr);
        } else if ((ch<BLANK||ch==0177) && iptr->evis) {
                eputc(UPARROW,iptr,cptr);
                eputc(ch+0100,iptr,cptr);       /* make it printable    */
        } else {
                eputc(ch,iptr,cptr);
        }
}

/*------------------------------------------------------------------------
 *  eputc - put one character in the echo queue
 *------------------------------------------------------------------------
 */
LOCAL eputc(ch,iptr,cptr)
        char    ch;
        struct  tty   *iptr;
        struct  csr    *cptr;
{
        iptr->ebuff[iptr->ehead++] = ch;
        if (iptr->ehead >= EBUFLEN)
                iptr->ehead = 0;
        cptr->ctstat = SLUENABLE;
}
```

12.8 tty Control block Initialization

Procedure `ttyinit` initializes the tty control block and interrupt vectors, given a pointer to the `devtab` structure for that device.

- It calls `iosetvec`: sets interrupt vector and fills in the interrupt dispatch table.
- It initializes the control block for `raw` or `cooked` mode, depending on whether or not the line corresponds to the console terminal.
- It creates input and output buffer semaphores and initialize `ibuf`, `obuf`, `ebuf`.
- After parameters, buffers, and interrupt vector have been set, it
  - clears receiver buffer in hardware,
  - enables recipient of interrupts and
  - disables transmitter interrupts.

Lower-Half Input Processing

```
/* ttyinit.c - ttyinit */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttyinit - initialize buffers and modes for a tty line
 *------------------------------------------------------------------------
 */
ttyinit(devptr)
        struct  devsw   *devptr;
{
        register struct tty *iptr;
        register struct csr *cptr;
        int      junk, isconsole;

        /* set up interrupt vector and interrupt dispatch table */

        iptr = &tty[devptr->dvminor];
        iosetvec(devptr->dvnum, iptr, iptr);
```

Lower-Half Input Processing

```
        devptr->dvioblk = iptr;                 /* fill tty control blk */
        isconsole = (devptr->dvnum == CONSOLE); /* make console cooked  */
        iptr->ioaddr = devptr->dvcsr;           /* copy in csr address  */
        iptr->ihead = iptr->itail = 0;          /* empty input queue    */
        iptr->isem = screate(0);                /* chars. read so far=0 */
        iptr->osem = screate(OBUFLEN);          /* buffer available=all */
        iptr->odsend = 0;                       /* sends delayed so far */
        iptr->ohead = iptr->otail = 0;          /* output queue empty   */
        iptr->ehead = iptr->etail = 0;          /* echo queue empty     */
        iptr->imode = (isconsole ? IMCOOKED : IMRAW);
        iptr->iecho = iptr->evis = isconsole;   /* echo console input   */
        iptr->ierase = iptr->ieback = isconsole;/* console honors erase */
        iptr->ierasec = BACKSP;                 /*  using ^h            */
        iptr->ecrlf = iptr->icrlf = isconsole;  /* map RETURN on input  */
        iptr->ocrlf = iptr->oflow = isconsole;  /* map RETURN on output */
        iptr->ikill = isconsole;                /* set line kill == @   */
        iptr->ikillc = ATSIGN;
        iptr->iintr = FALSE;
        iptr->ostart = STRTCH;
        iptr->ostop = STOPCH;
        iptr->icursor = 0;
        iptr->ifullc = TFULLC;
        cptr = (struct csr *)devptr->dvcsr;
        junk = cptr->crbuf;                     /* clear receiver and   */
        cptr->crstat = SLUENABLE;               /* enable in. interrupts*/
        cptr->ctstat = SLUDISABLE;              /* disable out.   "     */
}
```

12.9 Device Driver Control

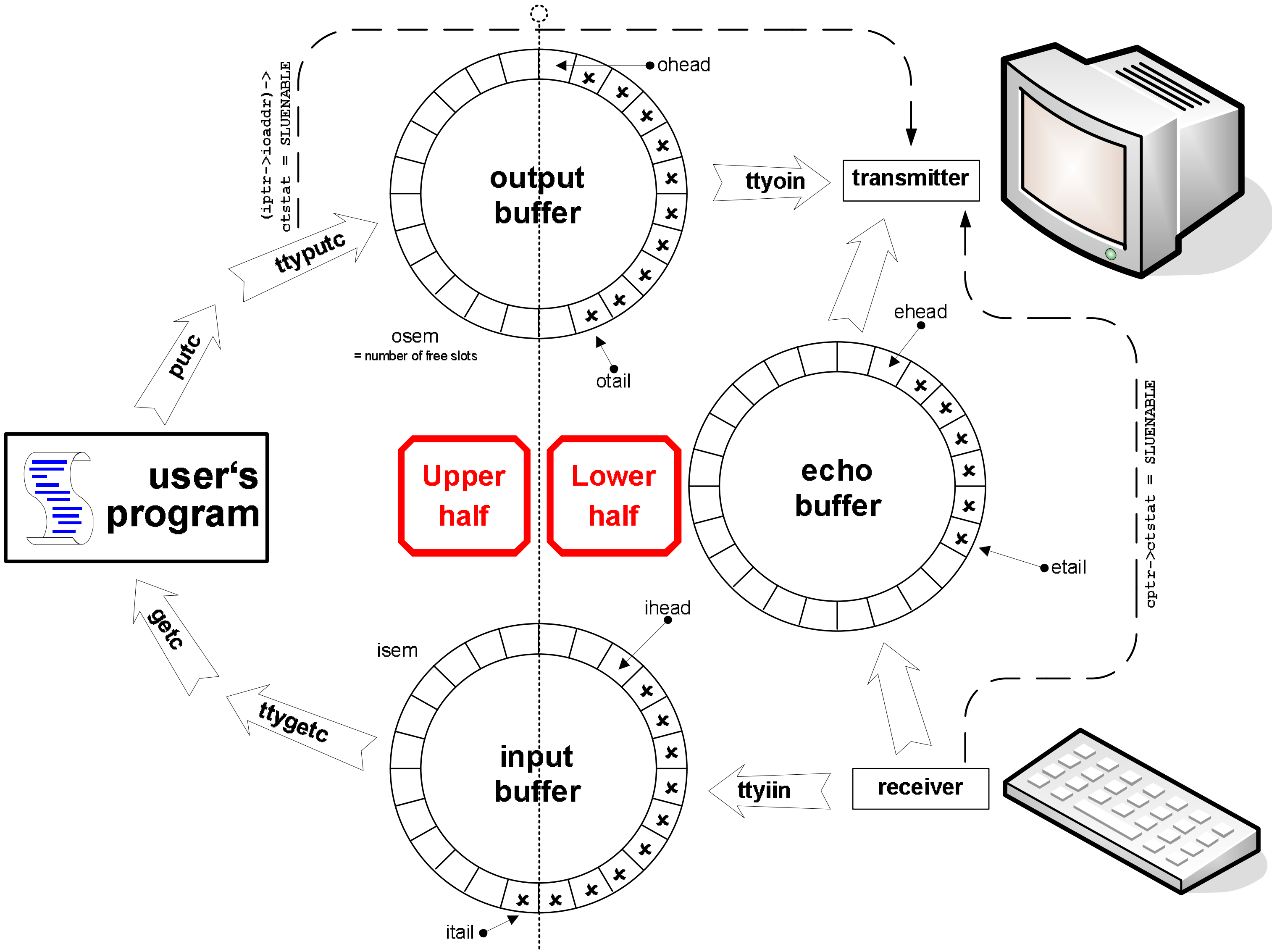
- `ttycntl` implements the higher-level device-independent I/O procedure `control(descrp,func,addr,addr2)` with `descrp` as an index into the device switch table and `func` setting the relevant mode.
- `ttycntl` controls a tty device by setting its modes, e.g.,
  - `func == TCSETBRK` sets the “break” bit in the SLU transmitter by placing the line in a break state,
  - `func == TCRSTBRK` turns off the break bit in the SLU transmitter and returns the line to normal processing,
  - `func == TCMODER` sets mode field `imode` to `raw`,
  - `func == TCMODEC` sets mode field `imode` to `cooked`,
  - `func == TCMODEK` sets mode field `imode` to `cbreak`,
  - `func == TCECHO` (`TCECHON`) control character echo
  - `func == TCICHARS` allows the user to query driver
  - `func =` ...

Device Driver Control

```
/* ttycntl.c - ttycntl */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <slu.h>

/*------------------------------------------------------------------------
 *  ttycntl  -  control a tty device by setting modes
 *------------------------------------------------------------------------
 */
ttycntl(devptr, func)
struct  devsw   *devptr;
int func;
{
        register struct tty *ttyp;
        char    ch;
        char    ps;

        ttyp = &tty[devptr->dvminor];
```

Device Driver Control

```
        switch ( func ) {
        case TCSETBRK:
                ttyp->ioaddr->ctstat |= SLUTBREAK;
                break;
        case TCRSTBRK:
                ttyp->ioaddr->ctstat &= ~SLUTBREAK;
                break;
        case TCNEXTC:
                disable(ps);
                wait(ttyp->isem);
                ch = ttyp->ibuff[ttyp->itail];
                restore(ps);
                signal(ttyp->isem);
                return(ch);
        case TCMODER:
                ttyp->imode = IMRAW;
                break;
        case TCMODEC:
                ttyp->imode = IMCOOKED;
                break;
        case TCMODEK:
                ttyp->imode = IMCBREAK;
                break;
        case TCECHO:
                ttyp->iecho = TRUE;
                break;
        case TCNOECHO:
                ttyp->iecho = FALSE;
                break;
        case TCICHARS:
                return(scount(ttyp->isem));
        default:
                return(SYSERR);
        }
        return(OK);
}
```


(iptr->ioaddr)->
ctstat = SLUENABLE
ohead
output
buffer
ttyoin
transmitter
ttyputc
putc
osem
= number of free slots
otail
ehead
user's
program
Upper
half
Lower
half
echo
buffer
etail
optr->ctstat = SLUENABLE
ihead
getc
isem
ttygetc
input
buffer
ttyiin
receiver
itail