13 Overview

- Device driver: a set of procedures controlling a peripheral hardware device
- Device driver routines partitioned into:
  - Upper-half device drivers: called from user programs
  - Lower-half device drivers: handling device interrupts
- These tow halves communicate via a shared data structure, *the device control block*
- In this chapter discussed: device driver for **tty**, managing:
  - output to display screen
  - input from keyboard

  using a SLU, an asynchronous serial line interface

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

A device driver for the CONSOLE **tty** introduces:

- The device switch table **devtab** forms the framework linking interrupts, devices and device driver routines
- The device driver routines for a standard computer terminal with a keyboard, called **teletype** - **tty**
- From the computer's viewpoint, this device consists of an serial asynchronous line unit ,SLU, between the keyboard and the system bus

13.1 The Device Type TTY

(1)

- Each **tty** has as input device a keyboard transmitting characters (witch are type) to the computer, and an output device for displaying characters received from the computer on its screen.
- The **tty** device driver consists of routines witch map operating system routines for reading/writing characters to display and receiving chars 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.

(2)

To minimize the interference between I/O devices and running processes, the driver uses interrupt- driven processing, to :

1. Transmit characters when SLU is idle,
2. read characters when these are received by the SLU,
3. handle errors in the receiving process (receiver),
4. coordinate requesters 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

(3)

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

- Severals parameters control the ëchoöf characters typed in 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 keystrokes automatically, and don't need the system to echo them
- Parameters concerning whether to echo unprintable control chars as printable combinations
- parameters dealing with moving to a new line:
  There are tow unprintable characters controlling movement of the cursor :
  - Return (also called "carriage return"): moving the cursor to the beginning of the current line
  - Newline : moving cursor vertically down by one line

(4)

However, programs like to deal with a single *end-of-line* character to move to a new line

In practice, terminal must receive both *newline* and *return* chars 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*
- **ocrlf :** controls mapping of *newline* to *move-to-the-start-of-a-new-line.*

13.2 Upper And Lower Halves of The Device Driver

(1)

The **tty** device drivers are partitioned in tow 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 from 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

(2)

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 action for output along the SLU.

I.e., there buffers record outgoing data from the time the user requests them to be sent until the time the device (SLU) receive them.

And record incoming data from the time the SLU deposits them until the user program requests them

(3)

These buffers are important for three reasons:

1. The driver can accept incoming characters before the user process reads them (since terminals user can start typing at any time)

2. Device s.a. disks transfer data in large blocks

3. These buffers permit the drivers to perform I/O concurrently with user processes

So the **tty** driver uses tow circular buffer per terminal - One for input

- One for output

Output:

- output operations issued by the calling user program are deposited as 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
- To pick up characters from the output buffer and places them into the output control area, thereby starting their transmission.

Input:

- Whenever the SLU receiver interrupts after/when it has received a characters, 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

13.2.1 Communication between the tow halves of the device driver

Ideally, the tow 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 tow halves do more than manipulate 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 that process until space becomes available.
- They start a process waiting for output from an empty buffer, as soon as the next character arrives.

13.3 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.
- But, there is a problem:
  - 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 by an any other list that the ready list:
    Output poses s 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.

13.4 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 implied by the above producer/consumer sinchronization for output op.

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 synchronization 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 op. in the upper-halve device driver has to *wait* for free space to become available, whenever the output op. in the lower-half *signals* that free space has become available, after an output op. in the SLU has interrupted, i.e. ended.
- In this way, the lower-half "Produce" free space, which upper-half "consumes"

13.5 Control Block and Buffer Declarations

(1)

- Each SLU used as **tty** has its own part of input and output buffer, and accompanying input and output semaphores

  isem(counting# of chars)

  osem(counting# of free buffer slots)

  This, and other info. is kept in the control block structure

- There is a control block structure for each **tty** device

- Control blocks also contains the control parameters

  i crlf

  o crlf

```
/* tty.h */

#define IOCHERR      0200  /* bit set in when an error    */
                           /* occurred reading the char   */
#define OBMINSP      20    /* min space in buffer before  */
                           /* processes awakend 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*/

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 eraising backspace on echo? */
        Bool evis;          /* echo control chars as ^X ?  */
        Bool ecrlf;         /* echo CR-LF for newline?     */
        Bool icrlf;         /* map '\r' to '\n' on inpute? */
        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 hedl?*/
        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[];

#define BACKSP  '\b'
#define BELL    '\07'
#define ATSING  '@'
#define BLANK   ''
#define NEWLINE '\n'
#define RETURN  '\r'
#define STOPCH  '\023'      /* control-S stops output      */
#define STRTCH  '\021'      /* control-Q restarts ouput    */
#define UPARROW '^'

/* ttycontrol function codes */

#define TCSETBARK  1      /* turn on BRAEK in transmitter */
#define TCRSTBARK  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*/
```

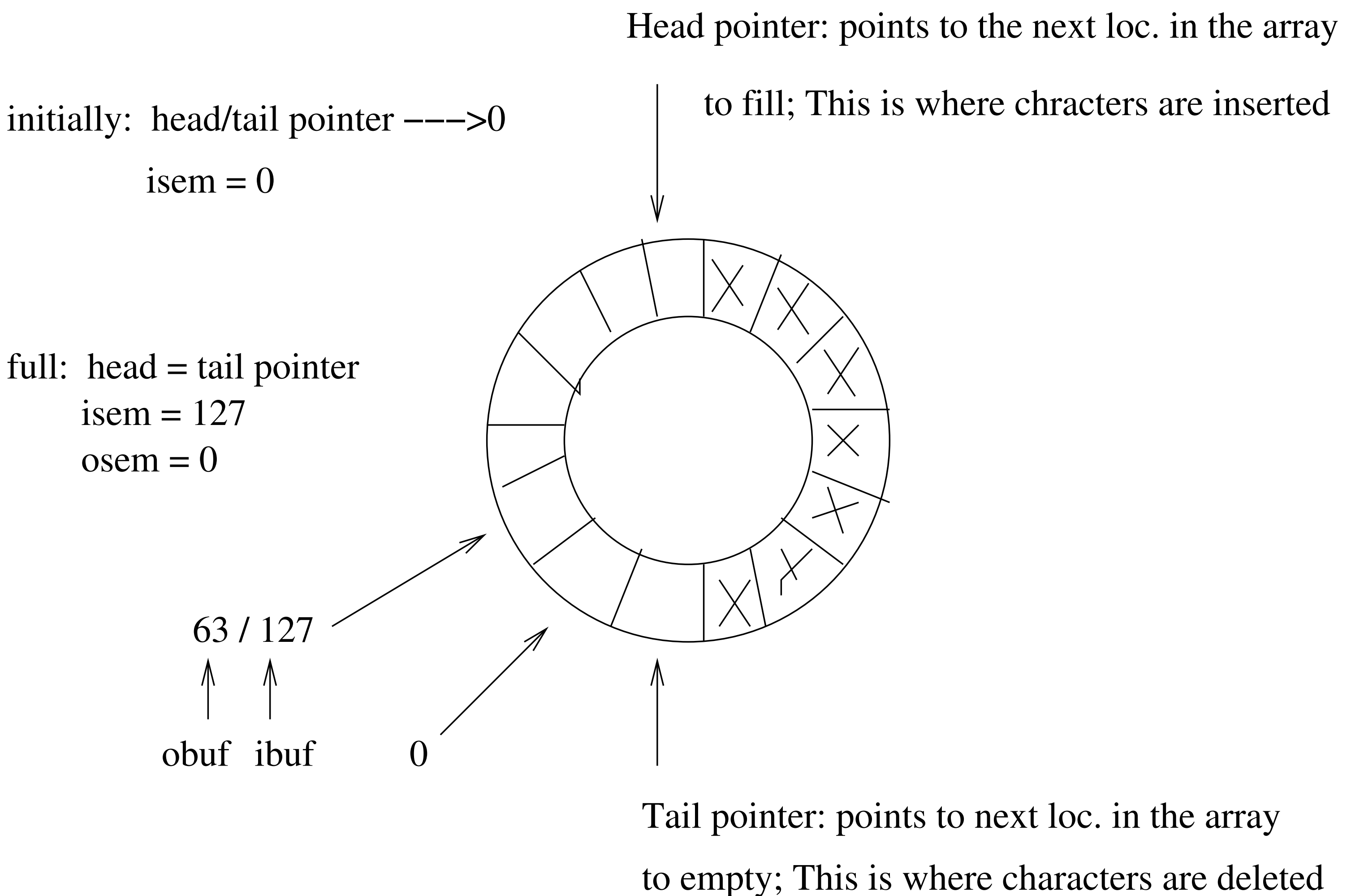
(2)

Key components:

input buffer *ibuff* + input semaphore *isem*

output buffer *o buff* + output semaphore *osem*

counting the number of free buffer slots- See the figure below :



(3)

- There is one control block structure per device in array **tty**, indexed by the minor device number.
- **Ntty**: number of **tty** device : the system config. program assign a number dev. nr. from 0 ... **Ntt-1** to each device.
- Also: system config. program places the device number dev. nr. in the dev. witch table **devtab**
- Both the interrupt device routines in lower half and driver routines in upper half use minor dev. numbers as index into array **tty**
- Control block contains the info. needed for synchronizing upper and lower half routines

13.6 Upper-half tty Input Routines

Routines :

ttygetc
ttyputc
ttyread
ttywrite

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

getc
putc
read
write

witch are high-level I/O ops. from chapter II

Simplest is ttygetc:

- *ttygetc* returns the minor dev. nr. from dev. sw. table and uses it as index into array **tty** to locate correct corresponding control block
- then it executes *wait(isem)* until lower-half input routine deposits a character in **ibuf**
- when *wait* returns, *ttygetc* extracts next char. from **ibuf** and updates tail pointer using modulo (128) arithmetic and returns.

```
#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;
        char   ch;
        struct tty  *iptr;

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

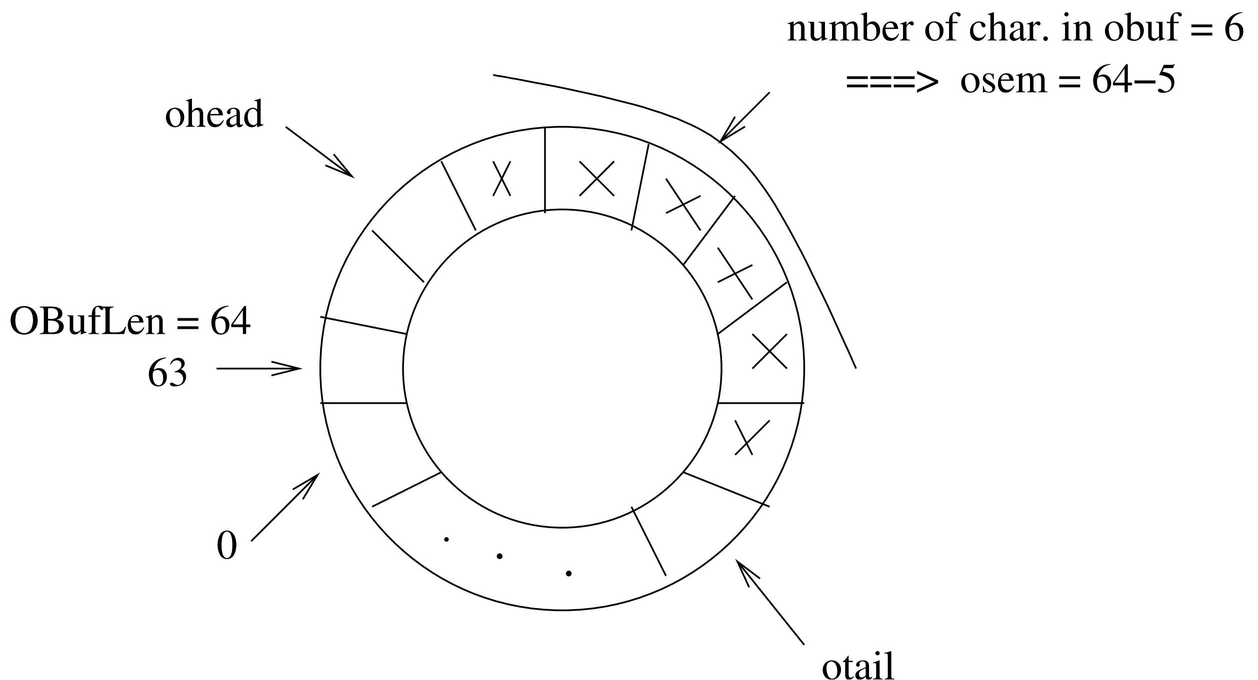
ttyread:

- It has three parameters:
  1. device switch table entry to identify min. dev. nr.
  2. buffer to which read input data should be output
  3. number of character to read **nread**
- reading all chars. in **ibuf** encoded by number of characters to read = 0
- tow cases treated :
  1. **bread** $\leq$ number of chars. in **ibuf**
  2. **nread** ¿ number of chars. in **ibuf**

```
/* ttyread.c - ttyread */

#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 *ptr;
        int avail, nread;

        if(count < 0)
              return(SYSERR);
        disable(ps);
```

```
        avail = scount( (iptr= &tty[devptr->dvminor])->isem );
        if ( (count = (count==0 ? avail : count)) =) {
                restore(ps;)
                return(0);
        }
        nread = count;
        if(count <= avail)
                readcopy(buff, iptr, count);
        else {
  if (avail > 0) {
                  readcopy(buff, iptr, avail);
                  buff += avail;
                  count -= avail;
          }
          for( ; count > 0; count--)
                 *buff++ = ttygetc(devptr);
        }
        restore(ps);
        return(nread);
}
```

13.7 Upper-Half tty Output Routines



- *ttyputc*: note **iptr** –¿**ocrlf** flag indicates whenever after ch == NEWLINE the return -char. 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 chars. inserted into **obuf**
- **ctstat**: identifies transmitter control and states field XCSR of the SLU-register.
  And bit I of XCSR set to 1 enables it interrupt when transmitter becomes idle.

- Structure *csr* defines the layout of the SLU device regs in the address space.
- To move data into SLU device reg., *ttyputc* needs to know the address assigned to SLU upon hardware installation.
- During device initialization this is copied into the *tty* control block field **ioaddr**, by calling *ttyinit* via calling *init(o)*
- *ttyinit* declared on later slide
- Using the device reg. 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 dev. output output interrupt enable bit is assigned to 1 by assigning SLUENABLE = 0100
- This is the only way upper-half routines to awaken low-half routines to initiate character transfer.

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

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

/*------------------------------------------------------
 * ttyputc - write one chracter 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);
    wait(iptr->osem);       /* wait for space in queue */
    disable(ps);
    iptr->obuff[iptr->ohead++] = ch;
    if (iptr->ohead >= OBUFLEN)
```

```
            iptr->ohead = 0;
    (iptr->iohead)->ctstat = SLUENABLE;
    restore(ps);
    return(OK);
}
```

```
/* 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   */
                         /* mask for error flages on input*/
#define SLUERMASK   0170000
#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) */
};
```

- SLUENABLE 0100 (octal)
- SLUREADY 0200 (octal)
- SLUERMASK 0170000 (octal)
- SLUCHMASK 0377 (octal)

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

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <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 avail;
         char ps;

         if(count < 0)
                return(SYSERR);
         disable(ps);
         ttyp = &tty[devptr->dvminor];
```

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

13.8 Lower-half tty driver routines

There are tow of them :

- lower-half input-interrupt routine *ttyiin*
- lower-half output-interrupt routine *ttyoin*

ttyoin: Code on later slide. Recall interrupt dispatcher calls *ttyoin* with interrupts disabled, wherever output device driver is idle

ttyoin(iptr) address of tty-control block for the interrupting device.

A *ttyoin(iptr)*-call considers 3 cases:

- a char. from echo buffer **ebuf** must be transmitted
- the driver should do something because output is currently interrupted until lines is full
- a character from the output buffer is transmitted.
  Tow ways to do this:
  - There is more than OBMINSP(=20) free space in **obuf**. Character is put in device reg. and *signal(osem)* called, to free on buffer slot
  - There is less than OBMINSP free space in **obuf**. Chars still transmitted. But *signal(osem)* deleted to halt *ttyputc*

```
#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <io.h>
#include <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) {
```

```
                cptr->ctstat = SLUISABLE;
                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;
}
```

13.9 Lower-Half Input Processing

Input interrupt processing moves complex because it caters for :

- character echo
- line editing
- processing of input errors
- buffer overflow

It operates in one of three modes, spec'd by *imode* field in *tty* control block :

- row : accumulates chars in **ibuff** without further processing
- cooked : dose character echo; honors suspend or restart output; accumulates full lines before passing them on to upper-half routines; honors input editing by erasing previous chars or killing entire lines
- cbreak : honors control chars but does no lines editing

In **raw** mode :

- *ttyiin* takes input char. from SLU-receiver unit
- deposits it in **ibuf**,
- signals **isem**
- if no space remains, *ttyiin* throws char. away errors reported by setting 8th-char. bit to 1

In **cbreak** mode, *ttyiin* takes input char. from SLU-receiver unit maps return to newline,

handles output flow control :

- field **oflow** determinate 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** control chars, 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-does character echo

In **cooked** mode, also performed : line editing,

- 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 char **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

```
#include <conf.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;
              }
                                      /* character error      */
              if ((ch=cptr->crbuf)&SLUERMASK)
                     iptr->ibuff[iptr->ihead++] =
                                          (ch&SLUCHMASK)| IOCHERR;
```

```
        else                        /* normal read complete */
                iptr->ibuff[iptr->ihead++] = ch & SLUCHMASK;
                                    /* wrap buffer pointer  */
        if (iptr->ihead >= IBUFLEN)
                iptr->ihead = 0;
        signal(iptr->isem);
} else {
        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 = FLASE;
                        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 {
        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;
        }
        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;
                                /* +1 for \n or \r */
                ct = iptr->icursor+1;
                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;
                }
        }
}
```

```
/*------------------------------------------------------------
 * 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,iptrcptr);
                          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;
}
```

```
/*------------------------------------------------------
 * 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;
}
```

13.10 Tty Control block Initialization

Procedure *ttyinit* initialize the tty control block and interrupt vectors, gives a ptr to the **devtab** structure for that device

- Calls *iosetvec*:
  sets interrupt and fills in interrupt dispatch table
- Initializes control block for **raw** or **cooked** mode, depending on whether or not line corresponds to the console terminal.
- Creates input and output buffer semaphores and initialize **ibuf, obuf, ebuf**
- After parameters, buffers, interrupt vectors have been set, it
  - clears receiver buffer in hardware,
  - enables recipient interrupt and
  - disable transmitter interrupts

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

/*------------------------------------------------------------
 * ttyinitialize buffer 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 vecotr and interrupt dispatch table */

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

      devptr->dvioblk = iptr;      /* fill tty control blk */
                                   /* make console cooked  */
      isconsole = (devptr->dvnum == CONSOLE);
      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 avaliable=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);
                                   /* echo console input */
    iptr->iecho = iptr->evis = isconsole;
                                 /* console honors erase */
    iptr->ierase = iptr->ieback = isconsole;
    iptr->ierasec = BACKSP;         /* using ^h           */
                                    /* map RETURN on input */
    iptr->ecrlf = iptr->icrlf = isconsole;
    iptr->ocrlf = iptr->oflow = isconsole;
    iptr->ikill = isconsole;        /* set line kill == @ */
    iptr->ikillc = ATSIGN;
    iptr->osheld = 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 = SLUISABLE;      /*disable out. interrupt*/
}
```

13.11 Device Driver Control

Ttycntl implements higher-level device independent I/O procedure

```
control(descrp,func,addr,addr2)
```

descrp: index into device switch table
func: set which mode ?

Ttycntl controls a **tty** device by setting its modes, e.g.,

- func == TCSETBRK : sets the ”break” bit in the SLU transmitter 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: raw
- func == TCMODEC: sets mode field imode: cooked
- func == TCMODEK: sets mode field imode: cbreak
- func == TCECHO (TCECHON): control chars. echo
- func == TCICHARS: allow the user to query driver
- func =

```
/* ttycntl.c - ttycntl */
/*------------------------------------------------------------------------
 * ttycntl - control a tty device by setting modes
 *------------------------------------------------------------------------
 */
ttycntl(devptr, func, addr)
struct  devsw  *devptr;
int func;
char *addr;
{
        register struct tty *ttyp;
        char     ch;
        char     ps;
        ttyp = &tty[devptr->dvminor];
        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);
}
```