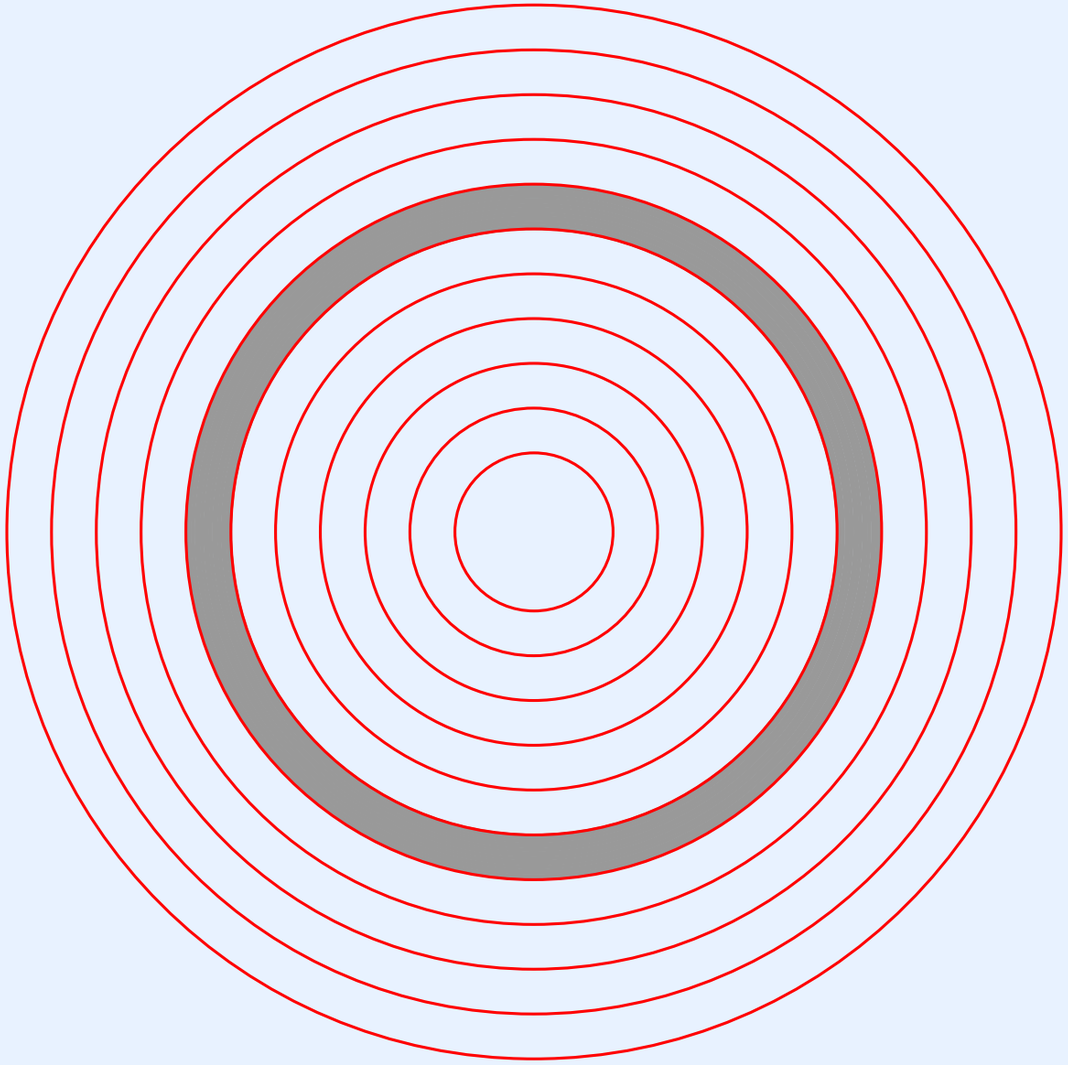
Module IX

Device Management
Device-Independent I / O
And An Example Device Driver

Location Of Device Management In The Hierarchy



Goals For A Devices Interface

- Isolation from hardware: ensure that applications do not contain details related to device hardware
- Portability: allow applications to run on any brand or model of equivalent device unchanged
- Elegance: limiting the interface to a small number of functions
- Generality: use a common paradigm across all devices
- Integration: integrate the device manager with the process manager and other operating system facilities

Achieving The Goals

- Devise a set of functions that applications use to
  - Obtain incoming data from a device
  - Transfer outgoing data to a device
  - Control the device
- Examples of control functions
  - Adjust the volume on headphones
  - Turn off character echo when reading a password
  - Eject a USB drive
- The approach is known as a *device-independent* interface

Achieving Device-independent I / O

- Define a set of abstract operations
- Build a function for each operation
- Have each function include an argument that a programmer can use to specify a particular device
- Arrange an efficient way to map generic operation onto code for a specific device

Xinu's Device-Independent I/O Primitives

- Follow the Unix open-read-write-close paradigm

init	–	initialize a device (invoked once, at system startup)
open	–	make a device ready for use
close	–	terminate use of a device
read	–	input arbitrary data from a device
write	–	output arbitrary data to a device
getc	–	input a single character from a device
putc	–	output a single character to a device
seek	–	position a device (primarily a disk)
control	–	control a device and/or its driver

- Note: some abstract functions may not apply to a given device

Implementation Of Device-Independent I/O In Xinu

- An application process
  - Makes calls to device-independent functions (e.g., *read*)
  - Supplies the device ID as parameter (e.g., ETHER)
- The device-independent I/O function
  - Uses the device ID to identify the correct hardware device
  - Invokes the appropriate device-specific function to perform the specified operation (e.g., *ethread* to read from an Ethernet)

Mapping A Generic I/O Function To A Device-Specific Function

- The mapping must be extremely efficient
- Solution: use a two-dimensional array
- The array is called a *device switch table*
  - A kernel data structure that is initialized when system loaded
  - Each row in the array corresponds to one device
  - Each column in the array corresponds to an operation
- An entry in the table points to a function to be called to perform the operation
- The device ID is chosen to be a index into rows of the table

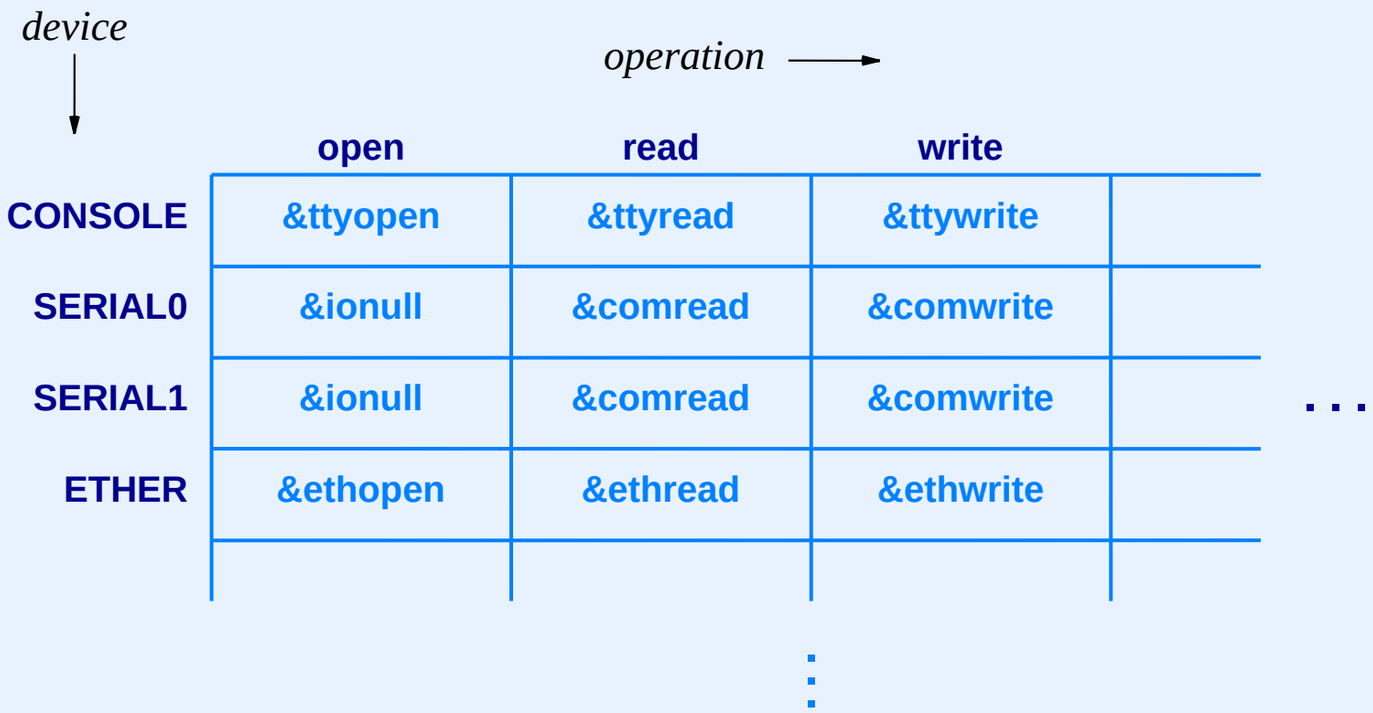
Entries In The Device Switch Table

- Each device-independent operation is generic
- However, a given operation may not make sense for a given device
  - *Seek* on keyboard, network, or display screen
  - *Close* on a mouse
- How should I/O functions handle the exceptions?
- To avoid special cases
  - Fill in *all* entries in the table
  - Place a valid function pointer in each entry
  - Create special functions for cases where an operation does not apply to a specific device

Special Entries Used In The Device Switch Table

- *ionull*
  - Used for an innocuous operation (e.g., *open* for a device that does not really require opening)
  - Simply returns *OK*
- *ioerr*
  - Used for an incorrect operation (e.g., *putc* on disk)
  - Simply returns *SYSERR*

Illustration Of Device Switch Table



- Each row corresponds to a device and each column corresponds to an operation
- An entry specifies the address of a function to invoke
- The example uses *ionull* for *open* on devices *SERIAL0* and *SERIAL1*

Replicated Devices And Device Drivers

- A computer may contain multiple copies of a given physical device
- Examples
  - Two Ethernet NICs
  - Two USB devices
- Goal have one copy of device driver code for the device and use the code with multiple devices

Parameterized Device Drivers

- A device driver must
  - Know which physical copy of a device to use
  - Keep information about the device separate from information for other copies
- To accommodate multiple copies of a device
  - Assign each instance a unique minor number (0, 1, 2, ...) known as its *minor device number*
  - Store the minor device number in the device switch table

Device Names

- Previous examples have shown examples of device names used in code (e.g., *CONSOLE*, *SERIAL0*, *SERIAL1*, *ETHER*)
- The device switch table is an array, and each device name is really an index into the array
- How does the system know how many rows to allocate in the table?
- How are unique values assigned to device names?
- How are minor device numbers assigned for replicated devices?
- Answer: a configuration program takes device information as input, including names to be used for devices, and generates the definitions and the device switch table entries automatically
- We will see more details later

Initializing The I/O Subsystem

- At system startup
  - Fill in the device switch table
  - Fill in interrupt vectors and initialize the interrupt hardware
  - Call *init* for each device, which initializes both the device hardware and the driver (e.g., creates the semaphores the driver uses for coordination)
- We will see more details later

An Example Device Driver

Our Example

- Consider a console device that connects to
  - A window on a user's screen
  - The user's keyboard
- The device is character-oriented
- Input consists of characters that come from the keyboard
- Output consists of characters sent to the screen
- Following the Unix convention, we used the term *tty* to describe the type of device

Hardware For The Example Device

- The underlying hardware consists of a *Universal Asynchronous Receiver and Transmitter* (*UART*)
- A UART transfers a single character (bytes) at a time, but the hardware has on-board input and output FIFOs
- When an input interrupt occurs
  - One or more characters are available in the input FIFO
  - The interrupt handler must extract all the characters
- Our driver maintains independent buffers for input and output
- Our driver uses semaphores to synchronize upper and lower halves

Tty Device Driver Functions

Upper-Half	Lower-Half
ttyinit	ttyhandler (interrupt handler)
ttyopen	ttyhandle_in (input interrupt)
ttyclose	ttyhandle_out (output interrupt)
ttyread	
ttywrite	
ttyputc	
ttygetc	
ttycontrol	

Actions Taken For Character Output

- An output semaphore counts spaces in the buffer
- When the upper-half is given a character
  - It waits on the output semaphore to guarantee buffer space is available
  - It deposits the character in next buffer slot
  - It "kicks" the device, which causes the device to interrupt
- The lower-half
  - Extracts the character from next filled slot in the buffer, and stores the character in the device output FIFO
  - It signals the semaphore to indicate that the buffer now has one more empty slot

Tty Driver Complexity

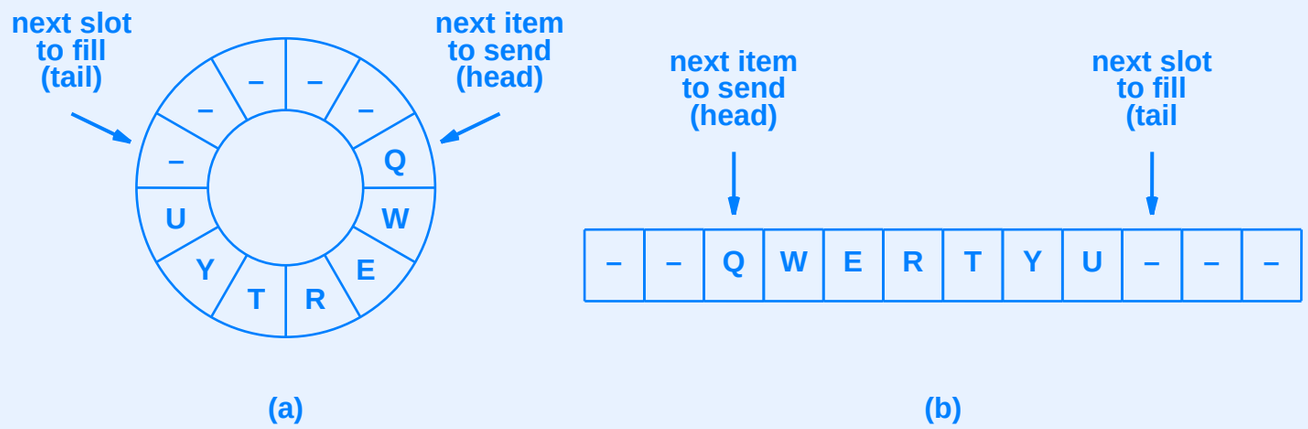
- The hardware is fairly “dumb”
- The driver software provides *modes*
  - Modes are similar to Unix that set many parameters (*cooked*, *cbreak*, *raw*)
- In addition, the driver allows many individual parameters to be controlled at any time
  - Whether CRLF mapping is in effect
  - Whether input character echo is turned on
  - Whether flow control (^S /^Q) is enabled
  - Whether control characters are visualized (e.g., ^A)
  - Whether backspacing over a character “erases” the character from the display

Tty Modes

Mode	Meaning
raw	The driver delivers each incoming character as it arrives without echoing the character, buffering a line of text, performing translation, or controlling the output flow
cooked	The driver buffers input, echoes characters in a readable form, honors backspace and line kill, allows type-ahead, handles flow control, and delivers an entire line of text
cbreak	The driver handles character translation, echoing, and flow control, but instead of buffering an entire line of text, the driver delivers each incoming character as it arrives

- The mode determines how input characters are processed
- An application can change the mode at any time

A Circular Buffer Implemented With An Array



- The figure shows

  (a) A circular buffer

  (b) An implementation with an array using head and tail integers to indicate positions

Definitions Used By The Tty Driver (Part 1)

```
/* tty.h */

#define TY_OBMINSP      20              /* Min space in buffer before   */
                                        /*   processes awakened to write*/
#define TY_EBUFLEN      20              /* Size of echo queue           */

/* Size constants */

#ifndef Ntty
#define Ntty            1               /* Number of serial tty lines   */
#endif
#ifndef TY_IBUFLEN
#define TY_IBUFLEN      128             /* Num. chars in input queue    */
#endif
#ifndef TY_OBUFLEN
#define TY_OBUFLEN      64              /* Num. chars in output queue   */
#endif

/* Mode constants for input and output modes */

#define TY_IMRAW        'R'             /* Raw input mode => no edits   */
#define TY_IMCOOKED     'C'             /* Cooked mode => line editing  */
#define TY_IMCBREAK     'K'             /* Honor echo, etc, no line edit*/
#define TY_OMRAW        'R'             /* Raw output mode => no edits  */
```

Definitions Used By The Tty Driver (Part 2)

```
struct  ttycblk {                       /* Tty line control block       */
        char    *tyihead;               /* Next input char to read      */
        char    *tyitail;               /* Next slot for arriving char  */
        char    tyibuff[TY_IBUFLEN];    /* Input buffer (holds one line)*/
        sid32   tyisem;                 /* Input semaphore              */
        char    *tyohead;               /* Next output char to xmit     */
        char    *tyotail;               /* Next slot for outgoing char  */
        char    tyobuff[TY_OBUFLEN];    /* Output buffer                */
        sid32   tyosem;                 /* Output semaphore             */
        char    *tyehead;               /* Next echo char to xmit       */
        char    *tyetail;               /* Next slot to deposit echo ch */
        char    tyebuff[TY_EBUFLEN];    /* Echo buffer                  */
        char    tyimode;                /* Input mode raw/cbreak/cooked */
        bool8   tyiecho;                /* Is input echoed?             */
        bool8   tyieback;               /* Do erasing backspace on echo?*/
        bool8   tyevis;                 /* Echo control chars as ^X ?   */
        bool8   tyecrlf;                /* Echo CR-LF for newline?      */
        bool8   tyicrlf;                /* Map '\r' to '\n' on input?   */
        bool8   tyierase;               /* Honor erase character?       */
        char    tyierasec;              /* Primary erase character      */
        char    tyierasec2;             /* Alternate erase character    */
        bool8   tyeof;                  /* Honor EOF character?         */
        char    tyeofch;                /* EOF character (usually ^D)   */
        bool8   tyikill;                /* Honor line kill character?   */
        char    tyikillc;               /* Line kill character          */
```

Definitions Used By The Tty Driver (Part 3)

```
        int32   tyicursor;              /* Current cursor position      */
        bool8   tyoflow;                /* Honor ostop/ostart?          */
        bool8   tyoheld;                /* Output currently being held? */
        char    tyostop;                /* Character that stops output  */
        char    tyostart;               /* Character that starts output */
        bool8   tyocrlf;                /* Output CR/LF for LF ?        */
        char    tyifullc;               /* Char to send when input full */
};
extern  struct  ttycblk ttytab[];

/* Characters with meaning to the tty driver */

#define TY_BACKSP       '\b'            /* Backspace character          */
#define TY_BACKSP2      '\177'          /* Alternate backspace char.    */
#define TY_BELL         '\07'           /* Character for audible beep   */
#define TY_EOFCH        '\04'           /* Control-D is EOF on input    */
#define TY_BLANK        ' '             /* Blank                        */
#define TY_NEWLINE      '\n'            /* Newline == line feed         */
#define TY_RETURN       '\r'            /* Carriage return character    */
#define TY_STOPCH       '\023'          /* Control-S stops output       */
#define TY_STRTCH       '\021'          /* Control-Q restarts output    */
#define TY_KILLCH       '\025'          /* Control-U is line kill       */
#define TY_UPARROW      '^'             /* Used for control chars (^X)  */
#define TY_FULLCH       TY_BELL         /* Char to echo when buffer full*/
```

Definitions Used By The Tty Driver (Part 4)

```
/* Tty control function codes */

#define TC_NEXTC        3               /* Look ahead 1 character       */
#define TC_MODER        4               /* Set input mode to raw        */
#define TC_MODEC        5               /* Set input mode to cooked     */
#define TC_MODEK        6               /* Set input mode to cbreak     */
#define TC_ICHARS       8               /* Return number of input chars */
#define TC_ECHO         9               /* Turn on echo                 */
#define TC_NOECHO       10              /* Turn off echo                */
```

Driver Definitions

- Note the complexity of the definitions
- Conclusion: although a tty device seems straightforward, the parameters used to control character processing complicate the driver
- Now consider driver functions to transfer data, perform control functions, and handle interrupts

Xinu Ttyputc (Part 1))

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyputc  -  Write one character to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyputc(
        struct  dentry  *devptr,        /* Entry in device switch table */
        char    ch                      /* Character to write           */
        )
{
        struct  ttycblk *typtr;         /* Pointer to tty control block */

        typtr = &ttytab[devptr->dvminor];

        /* Handle output CRLF by sending CR first */

        if ( ch==TY_NEWLINE && typtr->tyocrlf ) {
                ttyputc(devptr, TY_RETURN);
        }
```

Xinu Ttyputc (Part 2)

```
        wait(typtr->tyosem);            /* Wait for space in queue */
        *typtr->tyotail++ = ch;

        /* Wrap around to beginning of buffer, if needed */

        if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
                typtr->tyotail = typtr->tyobuff;
        }

        /* Start output in case device is idle */

        ttykickout((struct uart_csreg *)devptr->dvcsr);

        return OK;
}
```

Xinu Ttygetc (Part 1)

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttygetc  -  Read one character from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttygetc(
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        char    ch;                     /* Character to return          */
        struct  ttycblk *typtr;         /* Pointer to ttytab entry      */

        typtr = &ttytab[devptr->dvminor];

        /* Wait for a character in the buffer and extract one character */

        wait(typtr->tyisem);
        ch = *typtr->tyihead++;

        /* Wrap around to beginning of buffer, if needed */

        if (typtr->tyihead >= &typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyihead = typtr->tyibuff;
        }
```

Xinu Ttygetc (Part 2)

```
        /* In cooked mode, check for the EOF character */

        if ( (typtr->tyimode == TY_IMCOOKED) && (typtr->tyeof) &&
             (ch == typtr->tyeofch) ) {
                return (devcall)EOF;
        }

        return (devcall)ch;
}
```

Xinu Ttywrite

```
/* ttywrite.c - ttywrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttywrite  -  Write character(s) to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttywrite(
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer of characters         */
          int32 count                   /* Count of character to write  */
        )
{
        /* Handle negative and zero counts */

        if (count < 0) {
                return SYSERR;
        } else if (count == 0){
                return OK;
        }

        /* Write count characters one at a time */

        for (; count>0 ; count--) {
                ttyputc(devptr, *buff++);
        }
        return OK;
}
```

Xinu Ttyread (Part 1)

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyread  -  Read character(s) from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyread(
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer of characters         */
          int32 count                   /* Count of character to read   */
        )
{
        struct  ttycblk *typtr;         /* Pointer to tty control block */
        int32   avail;                  /* Characters available in buff.*/
        int32   nread;                  /* Number of characters read    */
        int32   firstch;                /* First input character on line*/
        char    ch;                     /* Next input character         */

        if (count < 0) {
                return SYSERR;
        }
```

Xinu Ttyread (Part 2)

```
typtr= &ttytab[devptr->dvminor];
if (typtr->tyimode != TY_IMCOOKED) {

        /* For count of zero, return all available characters */

        if (count == 0) {
                avail = semcount(typtr->tyisem);
                if (avail == 0) {
                        return 0;
                } else {
                        count = avail;
                }
        }
        for (nread = 0; nread < count; nread++) {
                *buff++ = (char) ttygetc(devptr);
        }
        return nread;
}

/* Block until input arrives */

firstch = ttygetc(devptr);
```

Xinu Ttyread (Part 3)

```
        /* Check for End-Of-File */

        if (firstch == EOF) {
                return EOF;
        }

        /* Read up to a line */

        ch = (char) firstch;
        *buff++ = ch;
        nread = 1;
        while ( (nread < count) && (ch != TY_NEWLINE) &&
                        (ch != TY_RETURN) ) {
                ch = ttygetc(devptr);
                *buff++ = ch;
                nread++;
        }
        return nread;
}
```

Xinu Ttycontrol (Part 1)

```
/* ttycontrol.c - ttycontrol */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttycontrol  -  Control a tty device by setting modes
 *------------------------------------------------------------------------
 */
devcall ttycontrol(
          struct dentry *devptr,        /* Entry in device switch table */
          int32  func,                  /* Function to perform          */
          int32  arg1,                  /* Argument 1 for request       */
          int32  arg2                   /* Argument 2 for request       */
        )
{
        struct  ttycblk *typtr;         /* Pointer to tty control block */
        char    ch;                     /* Character for lookahead      */

        typtr = &ttytab[devptr->dvminor];
```

Xinu Ttycontrol (Part 2)

```
/* Process the request */

switch ( func ) {

case TC_NEXTC:
        wait(typtr->tyisem);
        ch = *typtr->tyitail;
        signal(typtr->tyisem);
        return (devcall)ch;

case TC_MODER:
        typtr->tyimode = TY_IMRAW;
        return (devcall)OK;

case TC_MODEC:
        typtr->tyimode = TY_IMCOOKED;
        return (devcall)OK;

case TC_MODEK:
        typtr->tyimode = TY_IMCBREAK;
        return (devcall)OK;

case TC_ICHARS:
        return(semcount(typtr->tyisem));
```

Xinu Ttycontrol (Part 3)

```
        case TC_ECHO:
                typtr->tyiecho = TRUE;
                return (devcall)OK;

        case TC_NOECHO:
                typtr->tyiecho = FALSE;
                return (devcall)OK;

        default:
                return (devcall)SYSERR;
        }
}
```

Xinu Tty Handler (Part 1)

```
/* ttyhandler.c - ttyhandler */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyhandler  -  Handle an interrupt for a tty (serial) device
 *------------------------------------------------------------------------
 */
void ttyhandler(void) {
        struct  dentry  *devptr;        /* Address of device control blk*/
        struct  ttycblk *typtr;         /* Pointer to ttytab entry      */
        struct  uart_csreg *csrptr;     /* Address of UART's CSR        */
        byte    iir = 0;                /* Interrupt identification     */
        byte    lsr = 0;                /* Line status                  */


        /* Get CSR address of the device (assume console for now) */

        devptr = (struct dentry *) &devtab[CONSOLE];
        csrptr = (struct uart_csreg *) devptr->dvcsr;

        /* Obtain a pointer to the tty control block */

        typtr = &ttytab[ devptr->dvminor ];
```

Xinu Tty Handler (Part 2)

```
/* Decode hardware interrupt request from UART device */

/* Check interrupt identification register */
iir = csrptr->iir;
if (iir & UART_IIR_IRQ) {
        return;
}

/* Decode the interrupt cause based upon the value extracted    */
/* from the UART interrupt identification register.  Clear      */
/* the interrupt source and perform the appropriate handling    */
/* to coordinate with the upper half of the driver              */

/* Decode the interrupt cause */

iir &= UART_IIR_IDMASK;         /* Mask off the interrupt ID */
switch (iir) {

    /* Receiver line status interrupt (error) */

    case UART_IIR_RLSI:
        return;
```

Xinu Tty Handler (Part 3)

```
/* Receiver data available or timed out */

case UART_IIR_RDA:
case UART_IIR_RTO:

    resched_cntl(DEFER_START);

    /* While chars avail. in UART buffer, call ttyinter_in  */

    while ( (csrptr->lsr & UART_LSR_DR) != 0) {
            ttyhandle_in(typtr, csrptr);
    }

    resched_cntl(DEFER_STOP);

    return;
```

Xinu Tty Handler (Part 4)

```
            /* Transmitter output FIFO is empty (i.e., ready for more)  */

            case UART_IIR_THRE:
                ttyhandle_out(typtr, csrptr);
                return;

            /* Modem status change (simply ignore) */

            case UART_IIR_MSC:
                return;
            }
}
```

Input Interrupt Handling

- Recall that when an input interrupt occurs
  - One or more characters have arrived at the device
  - The driver must drain all characters from the device
- If multiple processes are waiting for input, the driver cannot let any of them proceed until all characters have been extracted from the device
- Technique used: defer rescheduling while extracting characters

Xinu Ttyhandle_in (Part 1)

```
/* ttyhandle_in.c - ttyhandle_in, erase1, eputc, echoch */

#include <xinu.h>

local   void    erase1(struct ttycblk *, struct uart_csreg *);
local   void    echoch(char, struct ttycblk *, struct uart_csreg *);
local   void    eputc(char, struct ttycblk *, struct uart_csreg *);

/*------------------------------------------------------------------------
 *  ttyhandle_in  -  Handle one arriving char (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyhandle_in (
          struct ttycblk *typtr,        /* Pointer to ttytab entry      */
          struct uart_csreg *csrptr     /* Address of UART's CSR        */
        )
{
        char    ch;                     /* Next char from device        */
        int32   avail;                  /* Chars available in buffer    */

        ch = csrptr->buffer;

        /* Compute chars available */

        avail = semcount(typtr->tyisem);
        if (avail < 0) {                /* One or more processes waiting*/
                avail = 0;
        }
```

Xinu Ttyhandle_in (Part 2)

```
/* Handle raw mode */

if (typtr->tyimode == TY_IMRAW) {
        if (avail >= TY_IBUFLEN) { /* No space => ignore input  */
                return;
        }

        /* Place char in buffer with no editing */

        *typtr->tyitail++ = ch;

        /* Wrap buffer pointer  */

        if (typtr->tyitail >= &typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyitail = typtr->tyibuff;
        }

        /* Signal input semaphore and return */
        signal(typtr->tyisem);
        return;
}

/* Handle cooked and cbreak modes (common part) */

if ( (ch == TY_RETURN) && typtr->tyicrlf ) {
        ch = TY_NEWLINE;
}
```

Xinu Ttyhandle_in (Part 3)

```
/* If flow control is in effect, handle ^S and ^Q */

if (typtr->tyoflow) {
        if (ch == typtr->tyostart) {        /* ^Q starts output */
                typtr->tyoheld = FALSE;
                ttykickout(csrptr);
                return;
        } else if (ch == typtr->tyostop) {  /* ^S stops output  */
                typtr->tyoheld = TRUE;
                return;
        }
}

typtr->tyoheld = FALSE;         /* Any other char starts output */
```

Xinu Ttyhandle_in (Part 4)

```
if (typtr->tyimode == TY_IMCBREAK) {       /* Just cbreak mode  */

        /* If input buffer is full, send bell to user */

        if (avail >= TY_IBUFLEN) {
                eputc(typtr->tyifullc, typtr, csrptr);
        } else {        /* Input buffer has space for this char */
                *typtr->tyitail++ = ch;

                /* Wrap around buffer */

                if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
                        typtr->tyitail = typtr->tyibuff;
                }
                if (typtr->tyiecho) {   /* Are we echoing chars?*/
                        echoch(ch, typtr, csrptr);
                }
        }
        return;
```

Xinu Ttyhandle_in (Part 5)

```
} else {        /* Just cooked mode (see common code above) */

        /* Line kill character arrives - kill entire line */

        if (ch == typtr->tyikillc && typtr->tyikill) {
                typtr->tyitail -= typtr->tyicursor;
                if (typtr->tyitail < typtr->tyibuff) {
                        typtr->tyitail += TY_IBUFLEN;
                }
                typtr->tyicursor = 0;
                eputc(TY_RETURN, typtr, csrptr);
                eputc(TY_NEWLINE, typtr, csrptr);
                return;
        }

        /* Erase (backspace) character */

        if ( ((ch==typtr->tyierasec) || (ch==typtr->tyierasec2))
                                    && typtr->tyierase) {
                if (typtr->tyicursor > 0) {
                        typtr->tyicursor--;
                        erase1(typtr, csrptr);
                }
                return;
        }
```

Xinu Ttyhandle_in (Part 6)

```
/* End of line */

if ( (ch == TY_NEWLINE) || (ch == TY_RETURN) ) {
        if (typtr->tyiecho) {
                echoch(ch, typtr, csrptr);
        }
        *typtr->tyitail++ = ch;
        if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyitail = typtr->tyibuff;
        }
        /* Make entire line (plus \n or \r) available */
        signaln(typtr->tyisem, typtr->tyicursor + 1);
        typtr->tyicursor = 0;   /* Reset for next line  */
        return;
}

/* Character to be placed in buffer - send bell if      */
/*      buffer has overflowed                           */

avail = semcount(typtr->tyisem);
if (avail < 0) {
        avail = 0;
}
if ((avail + typtr->tyicursor) >= TY_IBUFLEN-1) {
        eputc(typtr->tyifullc, typtr, csrptr);
        return;
}
```

Xinu Ttyhandle_in (Part 7)

```
/* EOF character: recognize at beginning of line, but   */
/*      print and ignore otherwise.                     */

if (ch == typtr->tyeofch && typtr->tyeof) {
        if (typtr->tyiecho) {
                echoch(ch, typtr, csrptr);
        }
        if (typtr->tyicursor != 0) {
                return;
        }
        *typtr->tyitail++ = ch;
        signal(typtr->tyisem);
        return;
}


/* Echo the character */

if (typtr->tyiecho) {
        echoch(ch, typtr, csrptr);
}

/* Insert in the input buffer */

typtr->tyicursor++;
*typtr->tyitail++ = ch;
```

Xinu Ttyhandle_in (Part 8)

```
                /* Wrap around if needed */

                if (typtr->tyitail >= &typtr->tyibuff[TY_IBUFLEN]) {
                        typtr->tyitail = typtr->tyibuff;
                }
                return;
        }
}

/*------------------------------------------------------------------------
 *  erase1  -  Erase one character honoring erasing backspace
 *------------------------------------------------------------------------
 */
local   void    erase1(
          struct ttycblk        *typtr, /* Ptr to ttytab entry          */
          struct uart_csreg     *csrptr /* Address of UART's CSRs       */
        )
{
        char    ch;                     /* Character to erase           */

        if ( (--typtr->tyitail) < typtr->tyibuff) {
                typtr->tyitail += TY_IBUFLEN;
        }
```

Xinu Ttyhandle_in (Part 9)

```
        /* Pick up char to erase */

        ch = *typtr->tyitail;
        if (typtr->tyiecho) {                       /* Are we echoing?   */
                if (ch < TY_BLANK || ch == 0177) { /* Nonprintable      */
                        if (typtr->tyevis) {    /* Visual cntl chars */
                                eputc(TY_BACKSP, typtr, csrptr);
                                if (typtr->tyieback) { /* Erase char    */
                                        eputc(TY_BLANK, typtr, csrptr);
                                        eputc(TY_BACKSP, typtr, csrptr);
                                }
                        }
                        eputc(TY_BACKSP, typtr, csrptr);/* Bypass up arr*/
                        if (typtr->tyieback) {
                                eputc(TY_BLANK, typtr, csrptr);
                                eputc(TY_BACKSP, typtr, csrptr);
                        }
                } else {  /* A normal character that is printable       */
                        eputc(TY_BACKSP, typtr, csrptr);
                        if (typtr->tyieback) {  /* erase the character  */
                                eputc(TY_BLANK, typtr, csrptr);
                                eputc(TY_BACKSP, typtr, csrptr);
                        }
                }
        }
        return;
}
```

Xinu Ttyhandle_in (Part 10)

```
/*------------------------------------------------------------------------
 *  echoch  -  Echo a character with visual and output crlf options
 *------------------------------------------------------------------------
 */
local   void    echoch(
          char  ch,                     /* Character to echo            */
          struct ttycblk *typtr,        /* Ptr to ttytab entry          */
          struct uart_csreg *csrptr     /* Address of UART's CSRs       */
        )
{
        if ((ch==TY_NEWLINE || ch==TY_RETURN) && typtr->tyecrlf) {
                eputc(TY_RETURN, typtr, csrptr);
                eputc(TY_NEWLINE, typtr, csrptr);
        } else if ( (ch<TY_BLANK||ch==0177) && typtr->tyevis) {
                eputc(TY_UPARROW, typtr, csrptr);/* print ^x            */
                eputc(ch+0100, typtr, csrptr);  /* Make it printable    */
        } else {
                eputc(ch, typtr, csrptr);
        }
}
```

Xinu Ttyhandle_in (Part 11)

```
/*------------------------------------------------------------------------
 *  eputc  -  Put one character in the echo queue
 *------------------------------------------------------------------------
 */
local   void    eputc(
          char  ch,                     /* Character to echo            */
          struct ttycblk *typtr,        /* Ptr to ttytab entry          */
          struct uart_csreg *csrptr     /* Address of UART's CSRs       */
        )
{
        *typtr->tyetail++ = ch;

        /* Wrap around buffer, if needed */

        if (typtr->tyetail >= &typtr->tyebuff[TY_EBUFLEN]) {
                typtr->tyetail = typtr->tyebuff;
        }
        ttykickout(csrptr);
        return;
}
```

Kicking A Device

- We said that kicking a device causes the device to interrupt
- The technique simplifies device driver software
- Key idea
  - If hardware is idle, kicking it forces an interrupt
  - If hardware is currently busy, kicking it has no effect (an interrupt will occur as usual when the operation completes)
- The point: kicking avoids a race condition because the processor does not ask the device whether it is idle before kicking it

Xinu Ttykickout

```
/* ttykickout.c - ttykickout */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttykickout  -  "Kick" the hardware for a tty device, causing it to
 *                  generate an output interrupt (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttykickout(
         struct uart_csreg *csrptr      /* Address of UART's CSRs       */
        )
{
        /* Force the UART hardware generate an output interrupt */

        csrptr->ier = UART_IER_ERBFI | UART_IER_ETBEI;

        return;
}
```

Xinu Ttyhandle_out (Part 1)

```
/* ttyhandle_out.c - ttyhandle_out */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyhandle_out  -  Handle an output on a tty device by sending more
 *                    characters to the device FIFO (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyhandle_out(
         struct ttycblk *typtr,         /* Ptr to ttytab entry          */
         struct uart_csreg *csrptr      /* Address of UART's CSRs       */
        )
{

        int32   ochars;                 /* Number of output chars sent  */
                                        /*   to the UART                */
        int32   avail;                  /* Available chars in output buf*/
        int32   uspace;                 /* Space left in onboard UART   */
                                        /*   output FIFO                */
        byte    ier = 0;

        /* If output is currently held, simply ignore the call */

        if (typtr->tyoheld) {
                return;
        }
```

Xinu Ttyhandle_out (Part 2)

```
/* If echo and output queues empty, turn off interrupts */

if ( (typtr->tyehead == typtr->tyetail) &&
     (semcount(typtr->tyosem) >= TY_OBUFLEN) ) {
        ier = csrptr->ier;
        csrptr->ier = ier & ~UART_IER_ETBEI;
        return;
}

/* Initialize uspace to the size of the transmit FIFO */

uspace = UART_FIFO_SIZE;

/* While onboard FIFO is not full and the echo queue is */
/*   nonempty, xmit chars from the echo queue           */

while ( (uspace>0) &&  typtr->tyehead != typtr->tyetail) {
        csrptr->buffer = *typtr->tyehead++;
        if (typtr->tyehead >= &typtr->tyebuff[TY_EBUFLEN]) {
                typtr->tyehead = typtr->tyebuff;
        }
        uspace--;
}
```

Xinu Ttyhandle_out (Part 3)

```
        /* While onboard FIFO is not full and the output queue is       */
        /*   nonempty, transmit chars from the output queue             */

        ochars = 0;
        avail = TY_OBUFLEN - semcount(typtr->tyosem);
        while ( (uspace>0) &&  (avail > 0) ) {
                csrptr->buffer = *typtr->tyohead++;
                if (typtr->tyohead >= &typtr->tyobuff[TY_OBUFLEN]) {
                        typtr->tyohead = typtr->tyobuff;
                }
                avail--;
                uspace--;
                ochars++;
        }
        if (ochars > 0) {
                signaln(typtr->tyosem, ochars);
        }
        return;
}
```

Xinu Ttyinit (Part 1)

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

#include <xinu.h>

struct  ttycblk ttytab[Ntty];

/*------------------------------------------------------------------------
 *  ttyinit  -  Initialize buffers and modes for a tty line
 *------------------------------------------------------------------------
 */
devcall ttyinit(
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  ttycblk *typtr;         /* Pointer to ttytab entry      */
        struct  uart_csreg *uptr;       /* Address of UART's CSRs       */

        typtr = &ttytab[ devptr->dvminor ];

        /* Initialize values in the tty control block */

        typtr->tyihead = typtr->tyitail =       /* Set up input queue   */
                &typtr->tyibuff[0];             /*    as empty          */
        typtr->tyisem = semcreate(0);           /* Input semaphore      */
        typtr->tyohead = typtr->tyotail =       /* Set up output queue  */
                &typtr->tyobuff[0];             /*    as empty          */
```

Xinu Ttyinit (Part 2)

```
typtr->tyosem = semcreate(TY_OBUFLEN);  /* Output semaphore     */
typtr->tyehead = typtr->tyetail =       /* Set up echo queue    */
        &typtr->tyebuff[0];             /*    as empty          */
typtr->tyimode = TY_IMCOOKED;           /* Start in cooked mode */
typtr->tyiecho = TRUE;                  /* Echo console input   */
typtr->tyieback = TRUE;                 /* Honor erasing bksp   */
typtr->tyevis = TRUE;                   /* Visual control chars */
typtr->tyecrlf = TRUE;                  /* Echo CRLF for NEWLINE*/
typtr->tyicrlf = TRUE;                  /* Map CR to NEWLINE    */
typtr->tyierase = TRUE;                 /* Do erasing backspace */
typtr->tyierasec = TY_BACKSP;           /* Primary erase char   */
typtr->tyierasec2= TY_BACKSP2;          /* Alternate erase char */
typtr->tyeof = TRUE;                    /* Honor eof on input   */
typtr->tyeofch = TY_EOFCH;              /* End-of-file character*/
typtr->tyikill = TRUE;                  /* Allow line kill      */
typtr->tyikillc = TY_KILLCH;            /* Set line kill to ^U  */
typtr->tyicursor = 0;                   /* Start of input line  */
typtr->tyoflow = TRUE;                  /* Handle flow control  */
typtr->tyoheld = FALSE;                 /* Output not held      */
typtr->tyostop = TY_STOPCH;             /* Stop char is ^S      */
typtr->tyostart = TY_STRTCH;            /* Start char is ^Q     */
typtr->tyocrlf = TRUE;                  /* Send CRLF for NEWLINE*/
typtr->tyifullc = TY_FULLCH;            /* Send ^G when buffer  */
                                        /*   is full            */

/* Initialize the UART */

uptr = (struct uart_csreg *)devptr->dvcsr;
```

Xinu Ttyinit (Part 3)

```
        /* Set baud rate */
        uptr->lcr = UART_LCR_DLAB;
        uptr->dlm = 0x00;
        uptr->dll = 0x18;

        uptr->lcr = UART_LCR_8N1;       /* 8 bit char, No Parity, 1 Stop*/
        uptr->fcr = 0x00;               /* Disable FIFO for now         */

        /* Register the interrupt dispatcher for the tty device */

        set_evec( devptr->dvirq, (uint32)devptr->dvintr );

        /* Enable interrupts on the device: reset the transmit and      */
        /*   receive FIFOS, and set the interrupt trigger level         */

        uptr->fcr = UART_FCR_EFIFO | UART_FCR_RRESET |
                        UART_FCR_TRESET | UART_FCR_TRIG2;

        /* Start the device */

        ttykickout(uptr);
        return OK;
}
```

Perspective

- UART hardware is primitive
- The software, not the hardware, displays characters on the screen as the user enters them on a keyboard
- Most features of a tty driver, such as erasing backspace, are handled entirely by software
- Unlike abstractions covered earlier (e.g., semaphores), a basic tty driver is incredibly complex
- A driver has many parameters that control its operation
- Small details complicate the code

Summary

- The *device manager* in an operating system provides an interface that applications use to request I/O
- Device-independent I/O functions
  - Provide a uniform interface
  - Define generic operations that must be mapped to device-specific functions
- Xinu uses a device switch table to map a device-independent operation to the correct driver function
- A device driver
  - Consists of functions that applications call to perform I/O on the device
  - Also provides an interrupt handler for the device
- Dynamic parameters and other details complicate a tty driver

Questions?