11

Device Independent Input and Output

Operating systems control input and output (I/O) devices for three reasons. First, the hardware interface to most devices is relatively crude, requiring complex software to control and use them. The operating system hides these details in routines called *device drivers* through which programs transfer data and control the device. Second, devices are shared resources, protected and allocated by the operating system according to policies that make access fair and safe. Third, the operating system provides a consistent, uniform, and flexible interface to all devices, allowing users to write programs that reference devices by name and perform high-level operations without knowing about the machine configuration. This chapter begins by looking at how a set of high-level primitives might be selected and proceeds to describe the data structures used to relate them to specific devices.

The choice of abstract input and output operations is not easy because goals like flexibility, simplicity, and generality tend to dictate contradictory designs. Any design will be an iterative process in which the designer chooses a set of primitives, maps device operations into them, and revises the choice as problems arise. However, the design process can be organized into roughly three steps that can be followed without many iterations. First: generate a list of desirable properties. Second: derive a set of high-level primitives, and explain the purpose of each by giving their meaning with respect to specific classes of devices (e.g., terminals, disks, etc.). Third: build software that maps an abstract device onto some particular instance of that device.

11.1 Properties Of The Input And Output Interface

What properties should I/O systems exhibit? Perhaps the most important design issue is synchrony: should processes block while performing I/O operations, or should they continue executing and be notified (somehow) when the operation completes. I/O systems that allow processes to initiate an operation and then to continue execution are called *asynchronous*; they are useful in a concurrent programming environment, especially when the user wants to control the overlap of computation and I/O. Those systems that delay input operations until the data arrives and delay output operations until the data has been consumed are called *synchronous*. They follow the pattern established in most high-level languages, and users generally prefer them. Synchronous I/O systems guarantee that the user can depend on data immediately after an input operation and can change data immediately after an output operation. Because it is easier to use, and because it seems to suffice for most applications, our design will use synchronous I/O.

The format of data and size of transfers is another issue that affects I/O system design. The question to ask is: ‘‘Will data be transferred in blocks or bytes, and if in blocks, of what size and what form?’’ These questions are difficult to answer because some devices work with single bytes while others operate on blocks of data − it may depend on the hardware that the system uses. A general purpose system, one that may be connected to a variety of I/O devices in a variety of configurations, will probably need both single-byte transfers as well as block transfers. Thus, our design will include both.

Finally, issues like efficiency, generality, portability, and simplicity arise. These can be ignored for now, but may ultimately force changes in the design. We will see in Chapter 16, for example, that lower-levels of the disk I/O software operate asynchronously.

11.2 Abstract Operations

Once a basic set of properties has been developed, a set of abstract I/O operations can be derived. Experience with other systems may be as important in helping to choose abstract operations as anything else. There are nine abstract operations designed for PC-Xinu: *getc*, *putc*, *read*, *write*, *control*, *seek*, *open*, *close*, and *init*. Throughout the rest of the book, we will refer to them as the I/O *primitives*.

Each primitive has a meaning, loosely defined as follows. *Getc* and *putc* deal with single character transfers, receiving from the device or sending to it. *Read* and *write* do the same for one or more characters transferred to a contiguous block of memory. *Control* allows a user to control the device or the device driver, with *seek* being a special case of *control* that applies to randomly accessible storage devices. *Open* and *close* allow the user to inform the device that data transfer will begin or that it has ended. These may be useful, for example, to return the device to an idle state when it is not needed. Finally, *init* initializes the device and driver at system startup.

Consider, for example, how these abstract routines apply to the ‘‘console.’’ *Getc* reads the next character from the keyboard, and *putc* displays one character on the screen. *Write* displays several characters with one call, and *read* reads a specified number of characters (or all that have been typed, depending on its arguments). Finally, *control* allows the program to change parameters in the driver to control such things as whether the system echoes each character as it is typed.

11.3 Binding Abstract Operations To Real Devices

The system maps high-level I/O operations like those described above into calls to specific device drivers. In so doing, it hides hardware and device driver details (e.g., that the keyboard and video display are actually independent devices), making programs independent of the hardware configuration. In one sense, these high-level calls comprise the environment that the system presents to running programs – the programs only perceive peripheral devices through these abstract primitives.

In addition to mapping abstract I/O operations onto driver routines, the system must map abstract names like *CONSOLE* onto real devices. Each system has its own mapping scheme; uniformity is not the rule. Some require the programmer to know about devices when writing the program. Others require the command interpreter to link names used in the program with real devices. Still others perform the linkage dynamically, allowing a running process to change the correspondence.

As a general rule, the later a system binds the names of abstract devices and abstract operations to real devices and device drivers, the more flexible it is. Programs that have device addresses and driver calls bound at compile-time are obviously impractical because they must be changed whenever the hardware device or address is altered, no matter how minor the change. At the other extreme, programs that bind names late usually incur more computational overhead – they are not as practical in small systems. So, the essence of the design problem consists of synthesizing a binding mechanism that allows maximum flexibility within the required performance bounds.

This chapter suggests a design which compromises between late binding and efficiency in a manner typical of many existing systems. Coded into the system is the device name, a description of each abstract device, the device driver routines it uses, and the address of a real device to which it corresponds. The system must be altered and recompiled when a new device is added or when existing device addresses are modified. User’s programs, however, contain no direct calls to device drivers and no device addresses; they need not be recompiled as long as the abstract device descriptors do not change. As a consequence, simple programs that only execute I/O operations on the *CONSOLE* work on almost any PC-Xinu configuration, independent of the physical console device, its hardware interface, or its hardware address.

11.4 Binding I/O Calls To Device Drivers At Run-Time

At some point, routines like *read* must map abstract device descriptors like *CONSOLE* to device driver routines and specific device addresses. Both the technicalities of how the mapping is performed, as well as the source of information about which devices the system contains, are important.

In PC-Xinu, each abstract device is assigned an integer *device descriptor* when the system is configured. By convention, the *CONSOLE* device has the same device descriptor in all Xinu systems. In addition, each device is assigned a unique name, which is a string of up to 7 characters. Based on the results of configuration, device descriptors and names are bound into the system when it is compiled. The system need not be recompiled unless the configuration changes (e.g., a new device is added). Once the system has been configured, any number of programs can be compiled. These programs address devices by name, and the compiler is able to map names to the correct device descriptors based on the configuration information.

At run-time the program calls high-level I/O routines like *read* or *putc*, passing the device descriptor as an argument. The high-level I/O routines use the device descriptor as an index into a table called the *device switch table*. The device switch maps each device descriptor into a real device address and appropriate driver routines. The high-level routine then calls the driver to carry out the operation.

A look at the definition of the device switch table, *devtab*, should clarify the details. It can be found in file *conf.h*. Structure *devsw*, declared in the same file, defines the format of entries in the device switch table.

```
/* conf.h */
/* (GENERATED FILE; DO NOT EDIT) */

#define NULLPTR (char *)0

/* Device table declarations */
struct  devsw   {                       /* device table entry           */
        int     dvnum;
        char    dvnam[10];
        int     (*dvinit)();
        int     (*dvopen)();
        int     (*dvclose)();
        int     (*dvread)();
        int     (*dvwrite)();
        int     (*dvseek)();
        int     (*dvgetc)();
        int     (*dvputc)();
        int     (*dvcntl)();
        int     dvport;
        int     dvivec;
```

```
        int     dvovec;
        int     (*dviint)();
        int     (*dvoint)();
        char    *dvioblk;
        int     dvminor;
        };

extern  struct  devsw devtab[];         /* one entry per device         */

/* Device name definitions */

#define CONSOLE 0               /* type tty     */
#define DS0     5               /* type dsk     */
#define DOS     11              /* type dos     */

/* Control block sizes */

#define Ntty    5
#define Ndsk    1
#define Ndf     5
#define Ndos    1
#define Nmf     4

#define NDEVS   16

/* Declarations of I/O routines referenced */

extern  int     ioerr();
extern  int     ttyinit();
extern  int     ttyopen();
extern  int     ttyread();
extern  int     ttywrite();
extern  int     ttygetc();
extern  int     ttyputc();
extern  int     ttycntl();
extern  int     ttyiin();
extern  int     lwinit();
extern  int     ionull();
extern  int     lwclose();
extern  int     lwread();
extern  int     lwwrite();
extern  int     lwgetc();
extern  int     lwputc();
extern  int     lwcntl();
```

```
extern  int     dsinit();
extern  int     dsopen();
extern  int     dsread();
extern  int     dswrite();
extern  int     dsseek();
extern  int     dscntl();
extern  int     lfinit();
extern  int     lfclose();
extern  int     lfread();
extern  int     lfwrite();
extern  int     lfseek();
extern  int     lfgetc();
extern  int     lfputc();
extern  int     msopen();
extern  int     mscntl();
extern  int     mfinit();
extern  int     mfclose();
extern  int     mfread();
extern  int     mfwrite();
extern  int     mfseek();
extern  int     mfgetc();
extern  int     mfputc();


/* Configuration and size constants */

#define MEMMARK                         /* enable memory marking        */
#define NPROC   30                      /* number of user processes     */
#define NSEM    100                     /* total number of semaphores   */

#define VERSION "6pc (1-Dec-87)"        /* label printed at startup     */
```

Each entry in *devtab* corresponds to a single device; it contains the device name, the addresses of the device driver routines for that device, the device port and vector addresses, and miscellaneous other information used by the drivers. Fields *dvgetc*, *dvputc*, *dvread*, *dvwrite*, *dvcntl*, *dvseek*, and *dvinit* hold the addresses of driver routines corresponding to the high-level operations. Knowing the addresses of driver routines is not enough, however, because more than one device can use the same driver routine. So, the device switch table contains fields for the hardware port address (*dvport*), interrupt vector addresses (*dvivec* and *dvovec*), and interrupt dispatch routines (*dviint* and *dvoint*), as well as a buffer pointer (*dvioblk*), and an integer to distinguish among multiple copies of a device (*dvminor*). The minor device number is especially important for multiplexors that control a set of identical devices through a single hardware interface.

11.5 The Implementation Of High-Level I/O Operations

Because the device switch table isolates high-level I/O operations from underlying details, it allows high-level procedures to be completed before device drivers. One of the chief benefits of such a strategy is that it allows the designer to build and test subsets of the I/O system.

There is a procedure for each of the abstract operations *getc*, *putc*, *read*, etc. This section describes the C implementation of these high-level routines and shows how they call low-level device drivers indirectly through the device switch table. For example, the C code in file *read.c* implements the *read* operation.

```
/* read.c - read */

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

/*------------------------------------------------------------------------
 *  read  --  read one or more bytes from a device
 *------------------------------------------------------------------------
 */
read(descrp, buff, count)
int descrp, count;
char *buff;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvread)(devptr,buff,count) );
}
```

A program calls *read*, and passes as arguments the device descriptor, the address of a buffer into which data should be read, and a count of the number of characters to read. Procedure *read* uses the device descriptor *descrp*, as an index into *devtab*, and calls the driver routine given by field *dvread*. It passes the driver three arguments: the address of the *devtab* entry (*devptr*), the buffer address (*buff*) and a count of characters to read (*count*).

The remaining high-level routines operate in the same way as *read*. They are shown below.

```
/* control.c - control */

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

/*------------------------------------------------------------------------
 *  control  --  control a device (e.g., set the mode)
 *------------------------------------------------------------------------
 */
control(descrp, func, addr, addr2)
int descrp, func;
char *addr,*addr2;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvcntl)(devptr, func, addr, addr2) );
}


/* getc.c - getc */

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

/*------------------------------------------------------------------------
 *  getc  --  get one character from a device
 *------------------------------------------------------------------------
 */
getc(descrp)
int descrp;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvgetc)(devptr) );
}
```

```
/* init.c - init */

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

/*------------------------------------------------------------------------
 *  init  --  initialize a device
 *------------------------------------------------------------------------
 */
init(descrp)
int descrp;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvinit)(devptr) );
}


/* putc.c - putc */

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

/*------------------------------------------------------------------------
 *  putc  --  write a single character to a device
 *------------------------------------------------------------------------
 */
putc(descrp, ch)
int descrp;
char ch;
{
        struct  devsw   *devptr;

        if (isbaddev    (descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvputc)(devptr,ch) );
}
```

```
/* seek.c - seek */

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

/*------------------------------------------------------------------------
 *  seek  --  position a device (very common special case of control)
 *------------------------------------------------------------------------
 */
seek(descrp, pos)
int descrp;
long pos;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvseek)(devptr,pos) );
}


/* write.c - write */

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

/*------------------------------------------------------------------------
 *  write  --  write 1 or more bytes to a device
 *------------------------------------------------------------------------
 */
write(descrp, buff, count)
        int descrp, count;
        char *buff;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvwrite)(devptr,buff,count) );
}
```

11.6 Translating Device Names Into Descriptors

Since the high-level operations described above require integer device descriptors to identify devices, there must be a way to map the device name into its descriptor. We could have written the high-level operations to use device names rather than descriptors, but the overhead to look up the device name at each call would be prohibitive.

The procedure *getdev* provides a means to translate a device name into a descriptor. *Getdev* is passed the address of a string and returns the device descriptor of the corresponding device, or *SYSERR*, if no device is found with the corresponding name. *Getdev* will usually be called only once for a device accessed by a program; the remaining access to the device will be through its device descriptor.

```
/* getdev.c - getdev */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 *  getdev  --  get the device number from a character string name
 *------------------------------------------------------------------------
 */

int getdev(cp)
char *cp;
{
        int i;

        for ( i=0; i<NDEVS; i++ )
                if ( strcmp(cp,devtab[i].dvnam) == 0 )
                        return(i);
        return(SYSERR);
}
```

11.7 Opening And Closing Devices

Some disk devices require the programs to start them before performing a transfer operation and to stop them when the transfer completes. Although *control* can be used in such situations, it is sometimes helpful to have more meaningfully named procedures to start up and shut down a device. *Open* and *close* serve this purpose. The code is again similar to that of the other high-level I/O routines:

```
/* close.c - close */

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

/*------------------------------------------------------------------------
 *  close  --  close a device
 *------------------------------------------------------------------------
 */
close(descrp)
int descrp;
{
        struct  devsw   *devptr;

        if (isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvclose)(devptr));
}


/* open.c - open */

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

/*------------------------------------------------------------------------
 *  open  --  open a connection to a device/file (arg1 & arg2 optional)
 *------------------------------------------------------------------------
 */
open(descrp, arg1, arg2)
int     descrp;
char    *arg1, *arg2;
{
        struct  devsw   *devptr;

        if ( isbaddev(descrp) )
                return(SYSERR);
        devptr = &devtab[descrp];
        return( (*devptr->dvopen)(devptr, arg1, arg2) );
}
```

11.8 Null And Error Entries In Devtab

High level routines like *read* and *write* use the entries in *devtab* without checking to see that they are valid. Thus, a driver address must be supplied for every operation and every device, or catastrophe may result (e.g., branch to zero). However, not all combinations of operations and devices are meaningful. For example, *seek* is not an operation that can be performed on the ctrl-break device. How can such *devtab* entries be filled in?

Two routines, *ioerr* and *ionull*, serve to fill in otherwise empty entries of *devtab* . Procedure *ioerr* simply returns *SYSERR* whenever it is called; procedure *ionull* always returns *OK*. By convention, entries filled with *ioerr* should never be called; they signify an illegal operation. Entries for unnecessary, but otherwise innocuous operations (like *open* for the real-time clock device), point to procedure *ionull*. The code for these routines is trivial.

```
/* ioerr.c - ioerr */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 *  ioerr  --  return an error (used for "error" entries in devtab)
 *------------------------------------------------------------------------
 */
ioerr()
{
        return(SYSERR);
}


/* ionull.c - ionull */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 *  ionull  --  do nothing (used for "don't care" entries in devtab)
 *------------------------------------------------------------------------
 */
ionull()
{
        return(OK);
}
```

11.9 Initialization Of The I/O System

We have seen: how the hardware uses the address in an interrupt vector location to locate the interrupt dispatch routine; how the interrupt dispatch routines use the interrupt dispatch table, *intmap*, and the common interrupt-handling code, *intcom*, to locate and execute the appropriate high-level interrupt routine; and how I/O system calls like *read* use *devtab* to map device descriptors into drivers when programs perform I/O operations. The question that remains is how these tables and interrupt vectors are initialized in the first place.

Devtab contains the entire system I/O configuration and is used when the system is compiled. The values in *devtab* vary from configuration to configuration, but a sample can be found in file *conf.c*:

```
/* conf.c */
/* (GENERATED FILE: DO NOT EDIT) */

#include <conf.h>
#include <bios.h>

/* device independent I/O switch */

struct  devsw   devtab[NDEVS] = {

/*------------------------------------------------------------------------
 * Format of each entry is:
 *
 * device number, device name,
 * init, open, close,
 * read, write, seek,
 * getc, putc, cntl,
 * port addr, device input vector, device output vector,
 * input interrupt routine, output interrupt routine,
 * device i/o block, minor device number
 *------------------------------------------------------------------------
 */

/* CONSOLE is tty on BIOS */
0,"tty",
ttyinit,ttyopen,ioerr,
ttyread,ttywrite,ioerr,
ttygetc,ttyputc,ttycntl,
0,KBDVEC| BIOSFLG,0,
ttyiin,ioerr,
```

```
NULLPTR,0,

/* GENERIC is tty on WINDOW */
1,"",
lwinit,ionull,lwclose,
lwread,lwwrite,ioerr,
lwgetc,lwputc,lwcntl,
0,0,0,
ioerr,ioerr,
NULLPTR,1,

/* GENERIC is tty on WINDOW */
2,"",
lwinit,ionull,lwclose,
lwread,lwwrite,ioerr,
lwgetc,lwputc,lwcntl,
0,0,0,
ioerr,ioerr,
NULLPTR,2,

/* GENERIC is tty on WINDOW */
3,"",
lwinit,ionull,lwclose,
lwread,lwwrite,ioerr,
lwgetc,lwputc,lwcntl,
0,0,0,
ioerr,ioerr,
NULLPTR,3,

/* GENERIC is tty on WINDOW */
4,"",
lwinit,ionull,lwclose,
lwread,lwwrite,ioerr,
lwgetc,lwputc,lwcntl,
0,0,0,
ioerr,ioerr,
NULLPTR,4,

/* DS0 is dsk on BIOS */
5,"ds0",
dsinit,dsopen,ioerr,
dsread,dswrite,dsseek,
ioerr,ioerr,dscntl,
0,0,0,
```

```
ioerr,ioerr,
NULLPTR,0,

/* GENERIC is df on DSK */
6,"",
lfinit,ioerr,lfclose,
lfread,lfwrite,lfseek,
lfgetc,lfputc,ioerr,
0,0,0,
ioerr,ioerr,
NULLPTR,0,

/* GENERIC is df on DSK */
7,"",
lfinit,ioerr,lfclose,
lfread,lfwrite,lfseek,
lfgetc,lfputc,ioerr,
0,0,0,
ioerr,ioerr,
NULLPTR,1,

/* GENERIC is df on DSK */
8,"",
lfinit,ioerr,lfclose,
lfread,lfwrite,lfseek,
lfgetc,lfputc,ioerr,
0,0,0,
ioerr,ioerr,
NULLPTR,2,

/* GENERIC is df on DSK */
9,"",
lfinit,ioerr,lfclose,
lfread,lfwrite,lfseek,
lfgetc,lfputc,ioerr,
0,0,0,
ioerr,ioerr,
NULLPTR,3,

/* GENERIC is df on DSK */
10,"",
lfinit,ioerr,lfclose,
lfread,lfwrite,lfseek,
lfgetc,lfputc,ioerr,
```

```
0,0,0,
ioerr,ioerr,
NULLPTR,4,

/* DOS is dos on MSDOS */
11,"dos",
ionull,msopen,ioerr,
ioerr,ioerr,ioerr,
ioerr,ioerr,mscntl,
0,0,0,
ioerr,ioerr,
NULLPTR,0,

/* GENERIC is mf on DOS */
12,"",
mfinit,ioerr,mfclose,
mfread,mfwrite,mfseek,
mfgetc,mfputc,ioerr,
0,0,0,
ioerr,ioerr,
NULLPTR,0,

/* GENERIC is mf on DOS */
13,"",
mfinit,ioerr,mfclose,
mfread,mfwrite,mfseek,
mfgetc,mfputc,ioerr,
0,0,0,
ioerr,ioerr,
NULLPTR,1,

/* GENERIC is mf on DOS */
14,"",
mfinit,ioerr,mfclose,
mfread,mfwrite,mfseek,
mfgetc,mfputc,ioerr,
0,0,0,
ioerr,ioerr,
NULLPTR,2,

/* GENERIC is mf on DOS */
15,"",
mfinit,ioerr,mfclose,
mfread,mfwrite,mfseek,
```

```
mfgetc,mfputc,ioerr,
0,0,0,
ioerr,ioerr,
NULLPTR,3
};
```

11.10 Interrupt Vector Initialization

The interrupt vectors and interrupt dispatch table are initialized at run-time, based on information in *devtab*. Each nonzero vector entry in *devtab* (either *dvivec* or *dvovec*) has a corresponding entry in the *intmap* table. The *intmap* entries are filled in when the system is initialized, by calling the *mapinit* procedure. *Mapinit* is passed the interrupt type, the address of the new interrupt service routine to be installed in the *intmap* table, and the minor device number of the device. (The minor device number will be passed to the interrupt service routine at each interrupt call.) The interrupt flag, which was discussed in Chapter 9, occupies the high byte of the interrupt type. Note how the old interrupt service routine segment:offset address is extracted from the vector and saved in the *oldisr* positions in the *intmap* entry, and the address of the call to *intcom* – which is one byte after the beginning of the *intmap* table entry – is installed in the vector.

The *maprestore* procedure is called when PC-Xinu terminates. It undoes the work performed by *mapinit* by restoring all the device vectors in the *intmap* table to their saved states.

```
/* map.c - mapinit, maprestore */

#include <dos.h>
#include <conf.h>
#include <kernel.h>
#include <io.h>

/*------------------------------------------------------------------------
 *  mapinit  --  fill in an intmap table entry
 *------------------------------------------------------------------------
 */
int mapinit(vec,newisr,mdevno)
int vec;                                /* interrupt vector no.         */
int (*newisr)();                        /* addr. of new service routine */
int mdevno;                             /* minor device number          */
{
        int i;                          /* intmap entry                 */
        word    far *addr;              /* far address pointer          */
        struct  intmap far *imp;        /* pointers to intmap           */
        int     flag;                   /* upper byte of vector         */
```

```
        i = nmaps;
        if ( i >= NMAPS )
                return(SYSERR);
        nmaps++;
        imp = &sys_imp[i];              /* point to our intmap entry    */
        flag = (vec>>8) & 0xff;         /* pick up flag byte            */
        vec = vec & 0xff;               /* only low-order byte counts   */
        FP_SEG(addr) = 0;               /* interrupts are on page 0     */
        FP_OFF(addr) = vec * 4;         /* offset of this interrupt no. */

/* set up the input intmap entry */
        imp->iflag = flag;              /* deposit flag byte in iflag   */
        imp->oldisr_off = *addr;        /* offset                       */
        imp->oldisr_seg = *(addr + 1);  /* segment                      */

/* the following is highly machine dependent */
        *addr = FP_OFF(imp)+1;          /* point to call instruction    */
        *(addr+1) = FP_SEG(imp);        /* this code segment            */
        imp->newisr = newisr;           /* our input handler            */
        imp->mdevno = mdevno;           /* minor device no.             */
        imp->ivec = (char) vec;         /* interrupt vector             */
        return(OK);
}

/*------------------------------------------------------------------------
 *  maprestore  --  restore all old interrupt vectors from the intmap
 *------------------------------------------------------------------------
 */
int maprestore()
{
        int     i;                      /* intmap entry number          */
        word    far *addr;              /* far address pointer          */
        struct  intmap far *imp;        /* pointers to intmap           */

        if ( nmaps > NMAPS )
                nmaps = NMAPS;          /* just to be sure              */
        for ( i=0; i<nmaps; i++) {
                imp = &sys_imp[i];      /* point to this intmap entry   */
                if ( (int)(imp->newisr) == -1 )
                        continue;       /* if unused entry              */
                FP_SEG(addr) = 0;       /* interrupts are on page 0     */
                FP_OFF(addr) = imp->ivec * 4;   /* offset to the vector */
                *addr = imp->oldisr_off;        /* offset               */
```

```
                *(addr+1) = imp->oldisr_seg;    /* segment              */
        }
}
```

At the same time the system is building the *intmap* table, the device initialization routine *init(k)* is called for each device *k*. This initialization routine, thought of as part of the device driver, usually initializes any control blocks or buffers associated with the device. It may also test the device, enable device interrupts, or reset the hardware in other ways as required. Part of this initialization depends on the driver and the device, but because so many devices initialize the interrupt vectors and dispatch table, building a standard high-level initialization routine is worthwhile.

11.11 Summary

The operating system provides a high-level environment to user programs by hiding the details of peripheral devices under a layer of device-independent I/O routines. User programs access devices by name using the high-level operations *getc*, *putc*, *read*, *write*, *control*, *seek*, *open*, and *close*. In our design, the I/O system operates *synchronously*, delaying the calling process until data has been transferred.

To keep device information in user's programs independent of hardware devices and addresses, the system binds abstract names to integer device descriptors. It binds descriptors to specific devices at run-time using a device switch table. The device switch table contains one entry for each device; the entry includes information like the device's hardware address as well as the set of driver routines that control the device. High level I/O operations, like *read* or *write*, access the device switch table to determine the driver routine that performs the operation on the specified device. Individual drivers interpret these calls in a way meaningful to the particular device; if an operation makes no sense when applied to a particular device, the system calls a routine that returns an error code.

One field of the device switch table specifies an initialization routine that the system calls at startup. Usually this initialization fills in the device control block, initializes buffers, and carries out any device-specific activities.

FOR FURTHER STUDY

The ideas of blocking I/O, most of the general I/O primitives, and the device switch table are not new. Although pieces can be found in several systems, the set described here came mostly from UNIX (Ritchie and Thompson [1974]). Two earlier systems that contributed to these ideas are Multics (Corbato [1972]) and CTSS (Crisman [1965]).

EXERCISES

11.1 Identify the set of I/O operations available on various operating systems.

11.2 Find a system that uses *asynchronous* I/O, and identify the mechanism by which a running program is notified when the operation completes. Which system would you rather use?

11.3 There is a difference between the binding of device names to device descriptors and the binding of device descriptors (e.g., 0) to real hardware devices. Compare the two bindings in PC-Xinu with bindings in other operating systems.

11.4 Assume that in the course of debugging you begin to suspect that a program is incorrectly calling high-level routines like *open* and *seek* on devices for which they make no sense. Make a quick change to catch I/O errors, printing the process id of the offending process. (Do not recompile the I/O system calls until you have tried other approaches.)

11.5 In one version of Xinu, *ioinit*, *ionull*, and *ioerr* were bound together in the same object file, making it impossible to load one without the others. Explain why separating them is a good idea.

11.6 Write the inverse of *getdev*, i.e., a procedure which is passed a device descriptor and returns a pointer to name of the device.