# *11. I/O indipendente dal dispositivo*

I sistemi operativi controllano i dispositivi di input e di output (I/O) per tre ragioni. In primo luogo, l'interfaccia che l'hardware mette a disposizione verso la maggior parte dei dispositivi è abbastanza povera, ed è quindi necessario un software complesso per controllarli ed usarli. Il sistema operativo nasconde questi particolari in routine chiamate *driver dei dispositivi (device driver)*, attraverso i quali i programmi trasferiscono dati e controllano i dispositivi. In secondo luogo, i dispositivi sono risorse condivise, protette e assegnate dal sistema operativo secondo politiche che rendono l'accesso equo e sicuro. Infine, il sistema operativo fornisce un'interfaccia coerente, uniforme e flessibile verso tutti i dispositivi, permettendo agli utenti di scrivere programmi che fanno riferimento ai dispositivi tramite nomi ed effettuano operazioni ad alto livello senza conoscere la configurazione della macchina. Questo capitolo inizia analizzando come può essere scelto un insieme di funzioni primitive di alto livello, e procede descrivendo le strutture dati utilizzate per metterle in relazione con gli specifici dispositivi.

La scelta delle operazioni di input/output astratte non è facile perché è difficile realizzare nello stesso progetto obiettivi come flessibilità, semplicità e generalità, essendo questi in contrasto tra loro. Ogni progetto dovrebbe essere un procedimento iterativo nel quale i progettisti scelgono un insieme di primitive, fanno corrispondere ad esse le operazioni sui dispositivi, e riesaminano la scelta fatta in base ai problemi che si sono presentati. Tuttavia, la progettazione può essere organizzata in (circa, approssimativamente) tre fasi che possono essere eseguite senza molte ripetizioni. Per prima cosa, preparare l'elenco delle proprietà che dovrebbero avere. Quindi, ricavare un insieme di primitive di alto livello, e spiegare lo scopo di ognuna, dandone il significato in riferimento alle specifiche classi di dispositivi (p.es. terminali, dischi, ecc.). Infine: scrivere il programma che fa corrispondere un dispositivo astratto su alcuni casi particolari di quel dispositivo.

## 11.1 Proprietà dell'interfaccia di input/output

Quali proprietà deve possedere un sistema di I/O ? Probabilmente il più importante problema di implementazione è la sincronia: se cioè i processi debbano bloccarsi mentre eseguono operazioni di I/O, o se debbano continuare la loro esecuzione e venga loro notificato (da qualcuno) quando l'operazione finisce. I sistemi di I/O che permettono ai processi di iniziare un'operazione e quindi continuare l'esecuzione sono chiamati *asincroni*; essi sono utili negli ambienti di programmazione concorrente, specialmente quando l'utente vuole controllare la sovrapposizione tra calcolo e I/O. Quei sistemi che ritardano le operazioni di input finché i dati non sono arrivati e ritardano quelle di output finché i dati non sono stati consumati sono chiamati *sincroni*. Essi seguono i modelli presenti nella maggior parte dei linguaggi ad alto livello, e generalmente gli utenti li preferiscono. I sistemi di I/O sincroni garantiscono che l'utente può contare sui dati immediatamente dopo un'operazione di I/O e può cambiarli immediatamente dopo un'operazione di output. Poiché è più semplice da utilizzare, e poiché si vede che è sufficiente per la maggior parte delle operazioni, il nostro progetto userà un I/O sincrono.

Il formato dei dati e la dimensione dei trasferimenti è un altro problema che concerne il progetto di sistemi di I/O. La domanda da farsi è: "I dati verranno trasferiti in blocchi o in byte e, se in blocchi, di quale dimensione e struttura ?". E' difficile rispondere a queste domande perché alcuni dispositivi lavorano con singoli byte mentre altri operano su blocchi di dati - ciò può dipendere dall'hardware che il sistema usa. Un sistema di uso generale, che può essere dunque collegato a molti dispositivi di I/O in molte configurazioni, avrà probabilmente bisogno sia del trasferimento a singoli bye sia di quello a blocchi. Perciò, il nostro progetto li includerà entrambi.

Infine, si presentano problemi come efficienza, generalità, portabilità, e semplicità. Questi possono essere ignorati per ora, ma potrebbero alla fine forzare cambiamenti nel progetto. Vedremo nel Capitolo 16, per esempio, che i livelli più bassi del software di I/O dei dischi lavorano in modo asincrono

## 11.2 Operazioni astratte

Una volta che un insieme basilare di proprietà è stato sviluppato, può essere ricavato un insieme di operazioni astratte di I/O. Le esperienze con altri sistemi possono essere importanti nell'aiutare a scegliere operazioni astratte come nient'altro. Ci sono nove operazioni astratte sviluppate per PC-Xinu: *getc*, *putc*,

*read*, *write*, *control*, *seek*, *open*, *close*, e *init*. Per tutto il resto del libro, ci riferiremo ad esse come le *primitive di I/O*.

Ogni primitiva ha un significato, più o meno definito come segue. *Getc* e *putc* si occupano del trasferimento di singoli caratteri, ricevendoli dal dispositivo o inviandoli ad esso. *Read* e *write* fanno lo stesso per uno o più caratteri trasferiti ad un blocco contiguo di memoria. *Control* permette ad un utente di controllare il dispositivo o il driver del dispositivo, mentre *seek* è un caso particolare di *control* che si usa per i dispositivi di immagazzinamento dei dati ad accesso casuale. *Open* e *close* permettono agli utenti di comunicare al dispositivo che il trasferimento dei dati deve cominciare o che è terminato. Questo può essere utile, ad esempio, per riportare il dispositivo ad uno stato inattivo quando non è utilizzato. Infine, *init* inizializza il dispositivo e il relativo driver all'avvio del sistema.

Si consideri, ad esempio, come queste routine astratte si applichino alla "console". *Getc* legge il successivo carattere dalla tastiera, e *putc* mostra un carattere sullo schermo. *Write* mostra parecchi caratteri con una chiamata, e *read* legge uno ben preciso numero di caratteri (o tutti quelli che sono stati digitati, seconda i suoi argomenti). *Control*, infine, permette al programma di cambiare i parametri nel driver per controllare alcune cose come ad esempio se il sistema mostri ogni carattere così come è stato digitato.

## 11.3 Collegare le operazioni astratte ai dispostivi reali

Il sistema fa corrispondere le operazioni di I/O ad alto livello come quelle descritte sopra a chiamate agli specifici dispositivi. Nel fare questo, esso nasconde i dettagli dell'hardware e del driver del dispositivo (p.es. che la tastiera e il video sono in realtà dispositivi indipendenti, anche se visti come unico dispositivo *CONSOLE*), rendendo i programmi indipendenti dalla configurazione dell'hardware. In un certo senso, queste chiamate ad alto livello racchiudono l'ambiente che il sistema mostra ai programmi in esecuzione - i programmi vedono le periferiche solamente attraverso queste primitive astratte.

Oltre a far corrispondere le operazioni astratte di I/O alle routine dei driver, il sistema deve far corrispondere i nomi astratti come *CONSOLE* ai dispositivi reali. Ogni sistema ha il suo schema per le corrispondenze; l'uniformità non è la regola. Alcuni sistemi richiedono ai programmatori di essere a conoscenza dei dispositivi nello scrivere i programmi. Altri richiedono all'interprete dei comandi di collegare i nomi usati nei programmi con i dispositivi reali. Altri ancora effettuano un collegamento dinamico, permettendo ad un processo in esecuzione di cambiare la corrispondenza.

Come regola generale, tanto più tardi un sistema lega i nomi dei dispositivi astratti e le operazioni astratte ai dispositivi reali e relativi driver, tanto più è flessibile. I programmi che legano gli indirizzi dei dispositivi e la chiamate ai driver in fase di compilazione non sono ovviamente pratici perché devono essere cambiati ogni volta che i dispositivi o gli indirizzi vengono alterati, per quanto piccolo possa essere il cambiamento. All'estremo opposto, i programmi che legano i nomi più tardi solitamente incorrono in un maggiore costo computazionale - essi non sono pratici nei sistemi piccoli. Così, l'essenza del problema consiste nel trovare un meccanismo di collegamento che permetta la massima flessibilità nei limiti delle prestazioni richieste.

Questo capitolo propone un modello che giunge ad un compromesso tra collegamento tardivo (late binding) ed efficienza in un modo tipico di molti sistemi esistenti. Nel sistema è codificato il nome dei dispositivi, una descrizione di ogni dispositivo astratto, le routine del driver del dispositivo da usare, e l'indirizzo del dispositivo reale a cui esso corrisponde. Il sistema deve essere modificato e ricompilato quando un nuovo dispositivo viene aggiunto o uno esistente viene modificato. I programmi utente possono, tuttavia, non contengono chiamate dirette ai driver dei dispositivi, né indirizzi di dispositivi; non c'è bisogno di ricompilarli se non cambiano i descrittori dei dispositivi astratti. Come conseguenza, semplici programmi che eseguono solamente operazioni di I/O sulla *CONSOLE* lavorano su quasi tutte le configurazioni di PC-Xinu, indipendentemente da come sia realmente la consolle, la sua interfaccia o i suoi indirizzi hardware.

## 11.4 Collegare le chiamate di I/O ai driver dei dispositivi in fase di esecuzione

A un certo punto, le routine come *read* devono far corrispondere i descrittori dei dispositivi come *CONSOLE* alle routine dei driver e agli specifici indirizzi dei dispositivi. Sono importanti sia l'aspetto tecnico di come viene compiuta la corrispondenza, come pure l'origine delle informazioni su quali dispositivi contenga il sistema.

In PC-Xinu, ad ogni dispositivo astratto è assegnato un valore intero, detto *descrittore di dispositivo*, quando il sistema è configurato. Per convenzione, il dispositivo CONSOLE ha lo stessi descrittore in tutti i

sistemi Xinu. In aggiunta, ad ogni dispositivo è assegnato un nome unico, che è una stringa di al più 7 caratteri. Dipendenti dalla compilazione, i descrittori di dispositivo e i nomi sono fissati nel sistema quando viene compilato. Il sistema non ha bisogno di essere ricompilato, a meno che non cambi la configurazione (p.es. viene aggiunto un nuovo dispositivo). Una volta che il sistema è stato configurato, può essere compilato un qualsiasi numero di programmi. Questi programmi indirizzano i dispositivi tramite il nome, e il compilatore è in grado di far corrispondere i nomi al corretto descrittore di dispositivo basato sulle informazioni di configurazione.

In fase di esecuzione il programma chiama le routine di I/O di alto livello come *read* o *putc*, passando come argomento il descrittore del dispositivo. Le routine di I/O ad alto livello usano il descrittore come un indice nella tabella chiamata *tabella di scelta del dispositivo* (*device switch table*). Ogni elemento della tabella fa corrispondere ogni descrittore di dispositivo al reale indirizzo del dispositivo e alle appropriate routine del driver. Le routine ad alto livello quindi chiamano il driver per portare a termine l'operazione.

Uno sguardo alla definizione device switch table, *devtab*, dovrebbe chiarire questi particolari. Essa si trova nel file *conf.h*. La struttura *devsw*, dichiarata nello stesso file, definisce il formato dell'elemento nella device switch table.

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

#define     NULLPTR     (char *)0

/* Dichiarazione della tabella dei dispositivi */
struct      devsw {                 /* elemento della tabella dei dispositivi */
      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[];         /* un elemento per dispositivo */

/* Definizioni dei nomi dei dispositivi */

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

/* Dimensioni dei blocchi di controllo (control blocks) */

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

#define     NDEVS 16        /* numero dei dispositivi */
```

```
/* Dichiarazione delle routine di I/O a cui si fa riferimento */

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   lwopen();
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   lfopen();
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();


/* Configurazione e costanti di dimensione */

#define MEMMARK                     /* abilita il memory marking */
#define NPROC   30                  /* numero dei processi utente */
#define NSEM    100                 /* totale dei semafori */


#define VERSION "6pc (1-Dec-87)"    /* stringa mostrata all'avvio */
```

Ogni elemento di devtab corrisponde ad un singolo dispositivo; esso ne contiene il nome, l'indirizzo delle routine del relativo driver, gli indirizzi delle port e dei vettori (di interrupt), e varie altre informazioni usate dai driver. I campi *dvgetc*, *dvputc*, *dvread*, *dvwrite*, *dvcntl*, *dvseek*, e *dvinit* contengono gli indirizzi delle routine dei driver corrispondenti alle operazioni ad alto livello. Conoscere gli indirizzi delle routine dei driver non è tuttavia sufficiente, perché più dipositivi possono usare la stessa routine. Così, la device switch table contiene campi per l'indirizzo della porta hardware (*dvport*), l'indirizzo del vettore di interrupt (*dvivec* e *dvovec*), e routine per la gestione degli interrupt (*dviint* e *dvoint*), così come un puntatore ad un buffer (*dvioblk*), e un numero intero per distinguere tra i dispositivi dello stesso tipo (*dvminor*). Il numero di

dispositivo secondario (minor device number) è particolarmente importante per i multiplexor che controllano un insieme di dispositivi identici attraverso un'unica interfaccia hardware.

## 11.5 L'implementazione di operazioni di I/O ad alto livello

Poiché la device switch table isola le operazioni di I/O ad alto livello dai dettagli sottostanti, permette di completare la stesura delle procedure ad alto livello prima di quella dei driver dei dispositivi. Uno dei principali benefici di tale strategia è che permette ai progettisti di costruire e testare dei sottoinsiemi del sistema di I/O.

C'è una procedura per ognuna delle operazioni astratte *getc*, *putc*, *read*, ecc. Questa sezione descrive l'implementazione in C di queste routine ad alto livello e mostra come esse chiamino indirettamente i driver dei dispositivi, di basso livello, attraverso la device switch table. Per esempio, il codice C nel file *read.c* implementa l'operazione *read*.

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

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

/*------------------------------------------------------------------------
 *  read  --  legge uno o più byte dal dispositivo
 *------------------------------------------------------------------------
 */
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) );
}
```

Un programma chiama *read*, e passa come argomenti il descrittore del dispositivo, l'indirizzo del buffer nel quale memorizzare i dati letti, e il numero di caratteri da leggere. La procedura *read* usa il descrittore di dispositivo, *descrp*, come indice in *devtab*, e chiama la routine di driver data dal campo *dvread*. Essa passa al driver tre argomenti: l'indirizzo dell'elemento di devtab considerato (*devptr*), l'indirizzo del buffer (*buff*) e il numero di caratteri da leggere (*count*).

Le altre routine ad alto livello operano nello stesso modo di *read*. Esse sono mostrate sotto.

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

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

/*------------------------------------------------------------------------
 *  control  --  controlla un dispositivo (p.es. imposta le proprietà)
 *------------------------------------------------------------------------
 */
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  --  legge un carattere da un dispositivo
 *------------------------------------------------------------------------
 */
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  --  inizializza un dispositivo
 *------------------------------------------------------------------------
 */
init(descrp, flag)
int descrp, flag;
{
      struct devsw *devptr;

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


/* putc.c - putc */

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

/*------------------------------------------------------------------------
 *  putc  --  scrive un singolo carattere sul dispositivo
 *------------------------------------------------------------------------
 */
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  -- posiziona un dispositivo (caso speciale di controllo molto comune)
 *------------------------------------------------------------------------
 */
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  --  scrive uno o più caratteri su un dispositivo
 *------------------------------------------------------------------------
 */
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 Tradurre i nomi dei dispositivi nei corrispondenti descrittori

Dal momento che le operazioni di alto livello descritte sopra richiedono un valore intero come descrittore per identificare i dispositivi, ci deve essere un modo per far corrispondere al nome di un dispositivo il relativo descrittore. Avremmo potuto scrivere le operazioni ad alto livello in modo data usare i nome dei

dispositivi piuttosto che i descrittori, ma il costo aggiuntivo della ricerca del nome del dispositivo ad ogni chiamata sarebbe stato proibitivo.

La procedura *getdev* fornisce il modo di tradurre il nome di un dispositivo nel corrispondente descrittore. *Getdev* riceve come parametro l'indirizzo di una stringa e restituisce il descrittore del dispositivo avente quel nome, o *SYSERR*, se non viene trovato alcun dispositivo con quel nome. *Getdev* verrà solitamente chiamata una sola volta per ogni dispositivo a cui un programma accede; tutti gli altri accessi al dispositivo avverranno tramite il suo descrittore.

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

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

/*------------------------------------------------------------------------
 *  getdev  --  ricava il numero del dispositivo dal nome (stringa di caratteri)
 *------------------------------------------------------------------------
 */

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 Aprire e chiudere dispositivi

Alcuni dispositivi (dischi) richiedono ai programmi di metterli in moto prima di effettuare un'operazione di trasferimento e di fermarli quando il trasferimento termina. Sebbene *control* possa essere usato in queste situazioni, talvolta può essere utile avere più procedure, aventi nomi significativi, per avviare e disattivare un dispositivo. *Open* e *close* sono utili a questo scopo. Il codice è ancora simile a quello delle altre routine di I/O ad alto livello:

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

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

/*------------------------------------------------------------------------
 *  close  --  chiude un dispositivo
 *------------------------------------------------------------------------
 */
int 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  --  apre una connessione ad un dispositivo/file
 *            (i parametri 2 e 3 sono opzionali)
 *------------------------------------------------------------------------
 */
int 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 Elementi Null ed Error in Devtab

Le routine ad alto livello come *read* e *write* usano gli elementi di *devtab* senza controllare se essi siano validi. Così, l'indirizzo di un driver deve essere fornito per ogni operazione ed ogni dispositivo, o il risultato potrebbe essere catastrofico (p.es. ????????). Tuttavia, non tutte le combinazioni di operazioni e dispositivi sono significativo. Per esempio, *seek* non è un'operazione che può essere effettuata sul dispositivo ctrl-break. Come possono essere riempiti questi elementi di *devtab* ?

Due routine, *ioerr* e *ionull*, servono a riempire gli elementi di *devtab* altrimenti vuoti. La procedura *ioerr* semplicemente restituisce *SYSERR* ogni volta che viene chiamata; la procedura *ionull* restituisce sempre *OK*. Per convenzione, gli elementi di *devtab* riempiti con *ioerr* non dovrebbero mai essere chiamati; essi indicano un'operazione illegale. Gli elementi che indicano operazioni non necessarie, ma altrimenti innocue (come *open* per l'orologio real-time) puntano alla procedura *ionull*. Il codice per queste routine è banale.

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

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

/*------------------------------------------------------------------------
 *  ioerr  --  restituisce un errore (usata come elemento "errore" in devtab)
 *------------------------------------------------------------------------
 */
ioerr()
{
      return(SYSERR);
}


/* ionull.c - ionull */

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

/*------------------------------------------------------------------------
 *  ionull  --  non fa nulla (usato per gli elementi di devtab che indicano
 *              operazioni non valide ma la cui chiamata non è grave
 *------------------------------------------------------------------------
 */
ionull()
```

```
{
      return(OK);
}
```

## 11.9 Inizializzazione del sistema di I/O

Abbiamo visto: come l'hardware usa l'indirizzo in una locazione del vettore di interrupt per individuare la routine di gestione dell'interrupt (interrupt dispatch routine); di come la interrupt dispatch routine usi la interrupt dispatch table, *intmap*, e il codice comune di trattamento degli interrupt, *intcom*, per individuare ed eseguire l'appropriata routine di interrupt ad alto livello; e di come il sistema di I/O usi chiamate come *read* e *devtab* per far corrispondere i descrittori dei dispositivi ai driver quando i programmi effettuano operazioni di I/O. La questione che rimane da analizzare è come queste tabelle e vettori di interrupt vengano inizializzati per la prima volta.

*Devtab* contiene la configurazione dell'intero sistema di I/O ed è usato quando il sistema viene compilato. I valori in *devtab* variano da configurazione a configurazione, ma un esempio può essere trovato nel 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] = {

/*------------------------------------------------------------------------
 * Il formato di ogni elemento è
 *
 * numero del dispositivo, nome del dispositivo,
 * init, open, close,
 * read, write, seek,
 * getc, putc, cntl,
 * indirizzo della porta, vettore di input, vettore di output,
 * routine di interrupt di input, routine di interrupt output,
 * device I/O block, numero secondario del dispositivo
 *------------------------------------------------------------------------
 */

/* 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,lwopen,lwclose,
lwread,lwwrite,ioerr,
lwgetc,lwputc,lwcntl,
0,0,0,
ioerr,ioerr,
NULLPTR,1,

/* GENERIC is tty on WINDOW */
2,"",
lwinit,lwopen,lwclose,
```

```
lwread,lwwrite,ioerr,
lwgetc,lwputc,lwcntl,
0,0,0,
ioerr,ioerr,
NULLPTR,2,

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

/* GENERIC is tty on WINDOW */
4,"",
lwinit,lwopen,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 Inizializzazione del vettore degli interrupt

Il vettore degli interrupt e la interrupt dispatch table sono inizializzati in fase di esecuzione, basandosi sulla informazioni presenti in *devtab*. Ogni elemento di *devtab* che punta ad un vettore non nullo (*dvivec* o *dvovec*) ha un corrispondente elemento in *intmap*. Gli elementi di *intmap* sono riempiti quando il sistema viene inizializzato, tramite la chiamata alla procedura *mapinit*. Ad essa vengono passati il tipo dell'interrupt, l'indirizzo della nuova routine di servizio dell'interrupt che deve essere installata in *intmap*, e il numero secondario di dispositivo (quest'ultimo verrà passato alla routine di servizio dell'interrupt ad ogni chiamata dell'interrupt). Si noti che l'indirizzo segmento:offset della vecchia routine di servizio dell'interrupt viene estratto dal vettore e salvato nella posizione *oldisr* dell'elemento di *intmap*, e l'indirizzo della chiamata a *intcom* - che è un byte dopo l'inizio dell'elemento di *intmap* - è installato nel vettore.

La procedura *maprestore* viene chiamata quando PC-Xinu termina. Essa annulla il lavoro effettuato da mapinit, infatti ripristina gli elementi del vettore di interrupt che riguardano i dispositivi, riportandoli allo stato salvato.

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

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

/*------------------------------------------------------------------------
 *  mapinit  --  riempie un elemento della tabella intmap
 *------------------------------------------------------------------------
 */
int mapinit(vec,newisr,mdevno)
int vec;                    /* no. del vettore di interrupt    */
int (*newisr)();            /* indirizzo della nuova routine di servizio */
int mdevno;                 /* numero secondario di dispositivo */
{
      int i;                /* indice in intmap: rappresenta un elemento */
      word  far *addr;      /* puntatore ad un indirizzo far */
      struct intmap far *imp; /* puntatore a intmap */
      int   flag;           /* byte superiore del vettore */

      i = nmaps;
      if ( i >= NMAPS )
            return(SYSERR);
      nmaps++;
      imp = &sys_imp[i];          /* punta alla nostro elemento di intmap */
      flag = (vec>>8) & 0xff;     /* prende il byte di flag */
      vec = vec & 0xff;           /* importano solo i byte bassi */
      FP_SEG(addr) = 0;           /* gli interrupt sono nel segmento 0 */
      FP_OFF(addr) = vec * 4;     /* offset del numero di questo interrupt */

/* imposta l'elemento di intmap dato in input */
      imp->iflag = flag;               /* copia il byte di flag in iflag */
      imp->oldisr_off = *addr;         /* offset */
      imp->oldisr_seg = *(addr + 1);   /* segmento */

/* ciò che segue è altamente dipendente dalla macchina */
      *addr = FP_OFF(imp)+1;      /* punta all'istruzione di chiamata (???) */
      *(addr+1) = FP_SEG(imp);    /* questo segmento di codice */
      imp->newisr = newisr;       /* il nostro gestore dell'input */
      imp->mdevno = mdevno;       /* numero secondario di dispositivo */
      imp->ivec = (char) vec;     /* vettore di interrupt */
      return(OK);
}

/*------------------------------------------------------------------------
```

```
 *  maprestore  --  ripristina tutti i vecchi vettori di interrupt da intmap
 *-------------------------------------------------------------------------
 */
int maprestore()
{
      int i;                        /* numero dell'elemento di intmap */
      word    far *addr;            /* puntatore ad un indirizzo far */
      struct  intmap far *imp;      /* puntatore ad intmap */

      if ( nmaps > NMAPS )
            nmaps = NMAPS;          /* tanto per essere sicuri */
      for (i=0; i<nmaps; i++) {
            imp = &sys_imp[i];      /* punta a questo elemento di intmap */
            if ( (int)(imp->newisr) == -1 )
                  continue;         /* se l'elemento è inutilizzato */
            FP_SEG(addr) = 0;       /* gli interrupt sono nel segmento 0 */
            FP_OFF(addr) = imp->ivec * 4;       /* offset al vettore */
            *addr = imp->oldisr_off;            /* offset */
            *(addr+1) = imp->oldisr_seg;        /* segmento */
      }
}
```

Nello stesso tempo in cui il sistema costruisce la tabella *intmap*, viene chiamata la routine di inizializzazione del dispositivo *init(k)* per ogni dispositivo *k*. Questa routine di inizializzazione, considerata come una parte del driver, solitamente inizializza tutti i blocchi di controllo o i buffer associati al dispositivo. Essa potrebbe anche provare il dispositivo, attivarne gli interrupt, o effettuare il reset dell'hardware nei diversi modi in cui è richiesto. Parte di questa inizializzazione dipende dal driver e dal dispositivo, ma poiché così tanti dispositivi inizializzano i vettori di interrupt e la interrupt dispatch table, vale la pena costruire una routine di inizializzazione standard di alto livello.

## 11.11 Sommario

Il sistema operativo fornisce un ambiente ad alto livello ai programmi utente nascondendo i dettagli delle periferiche sotto uno strato di routine di I/O indipendenti dal dispositivo. I programmi utente accedono ai dispositivi tramite il nome usando le operazioni ad alto livello *getc*, *putc*, *read*, *write*, *control*, *seek*, *open*, e *close*. Nel nostro progetto, il sistema di I/O lavora in modo *sincrono*, ritardando il processo chiamante finché i dati non sono stati trasferiti

Per mantenere le informazioni sui dispositivi nei programmi utente indipendenti dai dispositivi hardware w relativi indirizzi, il sistema lega i nomi astratti a valori interi, i descrittori di dispositivo. Esso lega i descrittori agli specifici dispositivi in fase di esecuzione usando la device switch table. Questa contiene un elemento per ogni dispositivo; l'elemento della tabella contiene informazioni come l'indirizzo hardware del dispositivo così come l'insieme delle routine del driver che controllano il dispositivo. Le operazioni di I/O ad alto livello, come *read* o *write*, accedono alla device switch table per determinare la routine del driver che effettua l'operazione su quel particolare dispositivo. I singoli driver interpretano queste chiamate in modo significativo per il particolare dispositivo; se un'operazione non ha senso se applicata ad un particolare dispositivo, il sistema chiama una routine che restituisce un codice d'errore.

Un campo della device switch table specifica una routine di inizializzazione che il sistema chiama all'avvio. Solitamente questa inizializzazione riempie i blocchi di controllo del dispositivo, inizializza i buffer, e porta a termine qualsiasi attività specifica del driver.

## Per approfondire

L'idea di un I/O bloccante, la maggior parte delle primitive di I/O, e la device switch table non sono nuove. Sebbene parte possa essere trovata in molti sistemi, l'insieme qui descritto proviene per la maggior parte da Unix (Ritchie e Thompson [1974]). Due precedenti sistemi che hanno contribuito a queste idee sono Multics (Corbato [1972]) e CTSS (Crisman [1965]).