11 Device Independent Input and Output

- Operating system **control I/O devices** for three reasons:
  - The hardware interface to such devices is **crude**, requiring **complex software packets** for their control, called **device drivers**.
  - Device drivers are **shared resources**, which need to be protected and allocated in a fair and safe way.
  - A **uniform, flexible, high-level interface** should be provided, allowing users to write programs without knowing the **machine configuration**.
- Discussed in this chapter are the selection of a set of **machine-independent high-level I/O primitives** and data structures required to relate these primitives to specific devices
- **How are these primitives selected?**
  - By generating a list of desirable properties; delivering a set of high-level primitives, giving their meaning with relation to certain abstract (classes of) devices, terminal, disks, etc.
  - Build software mapping the abstract devices to instances of that device.

11.1 Properties of the I/O interface

- Should processes **block** while performing I/O operations? Or should they **continue executing** and be **notified** when the operation completes?
  - **Asynchronous operations** are useful for controlling overlap – i.e., more parallelism – of computation and I/O operations.
  - **Synchronous operations** delay input operations until data arrives and output operations until data has been consumed. Their advantage is that users can depend on data **immediately** after an input operation, and change data **immediately** after an output operation.
- Which **format** does data have, and what is the **size** of transferred data?
  - Single-byte transfer (teletype terminals, e.g., the console)
  - Block transfer (a block of many bytes, used for data transfer to/from disks).

11.2 Abstract operations

- `getc()` and `putc( , ch)` deal with **single-character transfer**:
  - `getc()` reads next character from keyboard
  - `putc( , ch)` displays one character on terminal
- `read` and `write` deal with transfer to/from **continuous blocks of memory**:
  - `read` reads a specified number of characters
  - `write` displays several characters within one call
- `control` allows control of the device (driver): e.g., whether the system **echoes** each character as it is typed on the keyboard
- `seek` applies only to **randomly accessible memory (RAM)** to search for a particular position
- `open` and `close` inform device (drivers) that data transfer **will begin or has ended** (applies to disk and file access)
- `init` **initializes** the device and device driver at system start up

11.3 Binding abstract operations to real devices

- The system **maps** these high-level I/O operations to **specific device drivers**: It hides the details of the hardware and the device drivers. E.g., that keyboard and display are **independent**, it makes programs **independent** of the particular hardware configuration.
- The high-level calls of these operations constitute the **environment which the system presents to running programs** – i.e., the programs only perceive the peripheral devices through these abstract calls, see, e.g., §11.5 on page 147.
- The system also maps abstract names, s.a. **console**, to real devices.

Binding abstract operations to real devices

- Coded into the system is a **description of each abstract device**: E.g., the device driver routines which it uses, and the address of the real device to which it corresponds. Compare, e.g., §2.1.9 and §2.1.10 on pages 28 ff.
- When a new device is added to the system, or, e.g., the device addresses are modified, the system must be **altered and recompiled**.
- However, since programs do not contain **direct calls** to these devices and their drivers, and **no device addresses**, programs don't need to be recompiled, as long as the **abstract device descriptors** do not change (e.g., adding graphics facilities would change them).

11.4 Binding I/O calls to device drivers at run-time

- Routines like `read` in the compiled code should map **abstract device descriptors**, s.a., console, to **device driver routines** and real device addresses.
- In XINU, each abstract device is assigned an **integer device descriptor** (0, ..., 8), at system configuration. E.g., the console has the same device descriptor in all XINU systems.
- After system configuration, device descriptors are bound into the system when it is **compiled**, and these are placed in a **library**. No recompilation is needed unless the system changes (e.g, a new device is added).
- At **run-time** (after compilation) the program calls high-level routines, s.a., `read(descriptor, ..., ...)`, and `putc(descriptor, ch)` having the device descriptor as argument.
- **Device switch table**: `devtab[0 ... 8]` contains a structure of `devsw` containing the device drivers for that abstract device, and device addresses.
- A **device descriptor** is an index into the **device switch table** which contains an entry for each value of the device descriptor; read pages 144 to 146.

Device switch table

- Each entry in the device switch table corresponds to a **single device**, containing:
  - `dvnum`, the corresponding entry into the **interrupt dispatch table** `intmap`.
  - Addresses of the **device driver routines** for that device: `dvgetc` `devputc`, `dvread`, `dvwrite`, `dvcntrl`, `dvseek`, `dvinit`. These routines hold addresses of **driver routines** corresponding to these high-level operations.
  - Device address and other information, since **more than one device** can use the same device driver.
- Device switch table also contains:
  - hardware device addresses `dvcsr`,
  - interrupt vector addresses `dvivec` and `dvovec`,
  - the interrupt routines for input (`dviint`) and output (`dvoint`),
  - buffer pointer `dvioblk`, and
  - an integer `dvminor` distinguishing among multiple copies of a device.

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    *dvname;
        int     (*dvinit)();
        int     (*dvopen)();
        int     (*dvclose)();
        int     (*dvread)();
        int     (*dvwrite)();
        int     (*dvseek)();
        int     (*dvgetc)();
        int     (*dvputc)();
        int     (*dvcntl)();
        int     dvcsr;
        int     divec;
        int     dvovec;
        int     (*dviint)();
        int     (*dvoint)();
        char    *dvioblk;
        int     dvminor;
        };

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

Device switch table

`conf.h` continued:

```
/* Device name definitions */

#define CONSOLE     0                   /* type tty      */
#define OTHER       1                   /* type tty      */
#define RINGOIN     2                   /* type dlc      */
#define RINGOOUT    3                   /* type dlc      */
#define DISK0       4                   /* type dsk      */
#define FILE1       5                   /* type df       */
#define FILE2       6                   /* type df       */
#define FILE3       7                   /* type df       */
#define FILE4       8                   /* type df       */

/* Control block sizes */

#define Ntty    2
#define Ndlc    2
#define Ndsk    1
#define Ndf     4

#define NDEVS   9

/* Configuration and Size Constants */

#define MEMMARK                         /* define if memory marking used*/
#define NNETS       1                   /* number of Xinu ring networks */
                                        /*  (remove if there are zero)  */
#define NPROC       10                  /* number of user processes     */
#define NSEM        50                  /* number of semaphores         */
#define RTCLOCK                         /* system has a real-time clock */
#define VERSION     "6.1b (05/22/84)"   /* label printed at startup     */
```

Device switch table

`conf.h` continued:

```
/* Declarations of I/O routines referenced */

extern  int    ttyinit();  /* open e.g. on a console                   */
extern  int    ionull();   /* signifies an unnecessary but innocuous op. */
extern  int    ttyread();
extern  int    ttywrite();
extern  int    ioerr();    /* signifies an illegal operation           */
extern  int    ttycntl();
extern  int    ttygetc();
extern  int    ttyputc();
extern  int    ttyiin();
extern  int    ttyoin();
extern  int    dclinit();
extern  int    dclread();
extern  int    dclwrite();
extern  int    dlccntl();
extern  int    dlcputc();
extern  int    dlciin();
extern  int    dlcoin();
extern  int    dsinit();
extern  int    dsopen();
extern  int    dsread();
extern  int    dswrite();
extern  int    dsseek();
extern  int    dscntl();
extern  int    dsinter(); /* disk interrupt routine */
extern  int    lfinit();
extern  int    lfclose();
extern  int    lfread();
extern  int    lfwrite();
extern  int    lfseek();
extern  int    lfgetc();
extern  int    lfputc();
```

11.5 Implementation of high-level I/O operations

- There is a procedure call in the object code of a program for each of the abstract operations `getc`, `put`, `read` etc. – e.g.,

  `read(descrp, buff, count)`

  – `descrp` denotes the device descriptor: an index into device switch table `devtab`;

  – `buff` is the address of buffer into which the data read should be written;

  – `count` is the number of characters to be read.

- `read(..., ..., ...)` calls the driver routine given by field `dvread` of the structure `devtab` corresponding entry to `descrp`.

Implementation of high-level I/O operations

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

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

/*------------------------------------------------------------------------
 *  read  -  read one or more bytes from a device
 *------------------------------------------------------------------------
 */
SYSCALL 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) );
}
```

Implementation of high-level I/O operations

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

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

/*------------------------------------------------------------------------
 *  control  -  control a device (e.g., set the mode)
 *------------------------------------------------------------------------
 */
SYSCALL 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) );
}
```

Implementation of high-level I/O operations

```
/* getc.c - getc */

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

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

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

Implementation of high-level I/O operations

```
/* 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) );
}
```

Implementation of high-level I/O operations

```
/* putc.c - putc */

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

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

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

Implementation of high-level I/O operations

```
/* seek.c seek */

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

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

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

Implementation of high-level I/O operations

```
/* write.c - write */

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

/*------------------------------------------------------------------------
 *  write  -  write 1 or more bytes to a device
 *------------------------------------------------------------------------
 */
SYSCALL 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 Opening and closing devices

- Disk devices require programs to **start them up** before a transfer operation, and to **stop them** after the transfer has completed: `open` and `close`.

```
/* open.c - open */

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

/*------------------------------------------------------------------------
 *  open  -  open a connection to a device/file (parameters 2 & 3 are optional)
 *------------------------------------------------------------------------
 */
SYSCALL open(descrp, nam, mode)
int     descrp;
char    *nam;
char    *mode;
{
        struct  devsw   *devptr;

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

Opening and closing devices

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

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

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

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

11.7 Null and error entries in `devtab`

- Some entries in the device table are **not meaningful for a particular device**. E.g., `open` and `close` do not apply to consoles. Calling there operations would result in an error for console. How should these entries be filled in? By using:
  - `ioerr` – returns `SYSERR` when called. This signifies an illegal operation.
  - `ionull` – returns `OK`. This signifies an unnecessary but, otherwise, innocuous operation.

Null and error entries in devtab

```
/* 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);
}
```

11.8 Initialization of the I/O system

- `devtab` is initialized at system configuration time, so it is completely filled in by the time the system is compiled – see pages 154 and 155:

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

#include <conf.h>

/* device independent I/O switch */

struct  devsw   devtab[NDEVS] = {

/*  Format of entries is:
device-number,
init, open, close,
read, write, seek,
getc, putc, cntl,
device-csr-address, input-vector, output-vector,
iint-handler, oint-handler, control-block, minor-device,
*/

/*  CONSOLE  */

0,
ttyinit, ttyopen, ionull,
ttyread, ttywrite, ioerr,
ttygetc, ttyputc, ttycntl,
0177560, 0060, 0064,
ttyiin, ttyoin, NULLPTR, 0,
```

Initialization of the I/O system

```
/*  OTHER */

1,
ttyinit, ionull, ionull,
ttyread, ttywrite, ioerr,
ttygetc, ttyputc, ttycntl,
0177560, 0300, 0304,
ttyiin, ttyoin, NULLPTR, 1,

/*  RINGOIN  */

2,
dlcinit, ioerr, ioerr,
dlcread, dlcwrite, ioerr,
ioerr, dlcputc, dlccntl,
0176510, 0310, 0314,
dlciin, dlcoin, NULLPTR, 0,

/*  RINGOOUT  */

3,
dlcinit, ioerr, ioerr,
dlcread, dlcwrite, ioerr,
ioerr, dlcputc, dlccntl,
0176510, 0320, 0324,
dlciin, dlcoin, NULLPTR, 1,

/*  DISK0  */

4,
dsinit, dsopen, ioerr,
dsread, dswrite, dsseek,
ioerr, ioerr, dlccntl,
0176510, 0134, 0134,
dsinter, dsinter, NULLPTR, 0,

/*  FILE1  */

5,
lfinit, ioerr, lfclose,
lfread, lfwrite, lfseek,
lfgetc, lfputc, ioerr,
0000000, 0000, 0000,
ioerr, ioerr, NULLPTR, 0,

/*  FILE2  */

6,
lfinit, ioerr, lfclose,
lfread, lfwrite, lfseek,
lfgetc, lfputc, ioerr,
0000000, 0000, 0000,
ioerr, ioerr, NULLPTR, 1,

/*  FILE3  */

7,
lfinit, ioerr, lfclose,
lfread, lfwrite, lfseek,
lfgetc, lfputc, ioerr,
0000000, 0000, 0000,
ioerr, ioerr, NULLPTR, 2,

/*  FILE4  */

8,
lfinit, ioerr, lfclose,
lfread, lfwrite, lfseek,
lfgetc, lfputc, ioerr,
0000000, 0000, 0000,
ioerr, ioerr, NULLPTR, 3
        };
```

11.9 Interrupt vector initialization

- **Interrupt vectors** and the **interrupt dispatch table** `intmap` are **initalized at runtime**, using the information in `devtab`, by calling `init(0)`, ..., `init(8)` (defined on page 149). This is done in §13, see page 197. E.g., `ttyinit` calls

$$\texttt{iosetvec(descrp, }\underbrace{\texttt{incode}}_{\texttt{minor}}\texttt{, }\underbrace{\texttt{outcode}}_{\texttt{minor}}\texttt{)}$$

  where `minor := devtab[descrp.dvminor]` in `ioinit(descrp)`.

- At startup, for each device descriptor the system call `init(k)`:
  - initializes `dvinit` routine,
  - initializes **interrupt vectors** and fills in entries in **interrupt dispatch table** `intmap` (used by interrupt dispatchers to locate the high-level IO interrupt handlers for that device); this is done by calling `iosetvec(descrp, minor, minor)`,
  - initializes the control blocks (see §12) and buffers associated with that device.

Interrupt vector initialization

- This initialization occurs in every device initialization routine. Part of it is collected in the C-procedure `ioinit(descr)`, which calls `iosetvec(descr, minor, minor)`.
- `iosetvec` initializes the `intmap` structure for that device and fills in **input and output interrupt vectors**.
- The `devtab` table forms the **general framework** linking **interrupts**, **devices** and **device driver routines**.

Interrupt vector initialization

```
/* ioinit.c - ioinit, iosetvec */

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

/*------------------------------------------------------------------------
 *  ioinit --  standard interrupt vector and dispatch initialization
 *------------------------------------------------------------------------
 */
ioinit(descrp)
int     descrp;
{
        int     minor;

        if (isbaddev(descrp) )
                return(SYSERR);
        minor = devtab[descrp].dvminor;
        iosetvec(descrp, minor, minor);
        return(OK);
}
```

Interrupt vector initialization

ioinit.c continued:

```
/*------------------------------------------------------------------------
 *  iosetvec  -  fill in interrupt vectors and dispatch table entries
 *------------------------------------------------------------------------
 */
iosetvec(descrp, incode, outcode)
int     descrp;
int     incode;
int     outcode;
{
        struct  devsw   *devptr;
        struct  intmap  *map;
        struct  vector  *vptr;

        if (isbaddev(descrp))
                return(SYSERR);
        devptr = &devtab[descrp];
        map = &intmap[devptr->dvnum];   /* fill in interrupt dispatch   */
        map->iin =  devptr->dviint;     /*   map with addresses of high-*/
        map->icode = incode;            /*   level input and output     */
        map->iout = devptr->dvoint;     /*   interrupt handlers and     */
        map->ocode = outcode;           /*   minor device numbers       */
        vptr = (struct vector *)devptr->dvivec;
        vptr->vproc = (char *)INTVECI;  /* fill in address of input     */
                                        /* interrupt dispatch routine   */
        vptr->vps = descrp | DISABLE;   /*   vector PC and PS values    */
        vptr = (struct vector *)devptr->dvovec;
        vptr->vproc = (char *)INTVECO;  /* fill in address of output    */
                                        /* interrupt dispatch routine   */
        vptr->vps = descrp | DISABLE;   /*   vector PC and PS values    */
        return(OK);
}
```

11.10 Summary: Recap §11

- Call `read` in a program.
- Compilation results in translation into `read(descrp, buff, count)` – a XINU routine written in C.
- Executing that routine results in executing the `dvread` routine whose address is stored in the **device switch table** entry of `devtab`.
- That results in posting one (or more) **interrupts** to the device in question.
- Those interrupts result in:
  - storing the original *PC* and *PS* on the stack,
  - overwriting these registers with values from the **interrupt vector**, and
  - executing the **interrupt dispatch routines** contained in the interrupt vector.
- The interrupt vector contains the address of the **interrupt dispatch routine** for the device and **operations concerned** plus the **device identifier** (descriptor).

Summary: Recap §11

- After saving the values of those registers that will be used, the interrupt dispatch routine:
  - calls the appropriate **high-level I/O interrupt routines**
  - with appropriate arguments **identifying the device in question**.
- The I/O dispatch routine finds the **addresses of the interrupt handlers** in the interrupt dispatch table `intmap`, in which also the I/O device identifiers are stored.
- These addresses are obtained from the device table **at system initialization**. The I/O interrupt vectors and interrupt dispatch table are initialized through calling `init(0)`, ..., `init(8)`.
- Calling `init(descriptor)` results in caling the **device-dependent initialization routines**. The **standard** form of such a routine, `ioinit(descrp)`, is given on page 156, which calls `iosetvec`, part of every `dvinit` routine (see, e.g., `ttyinit` on page 182).