Sheets for MIEIC's SOPE

based on teaching material supplied by A. Tanenbaum for book: Modern Operating Systems, ed...

Chap 5: Input / Output

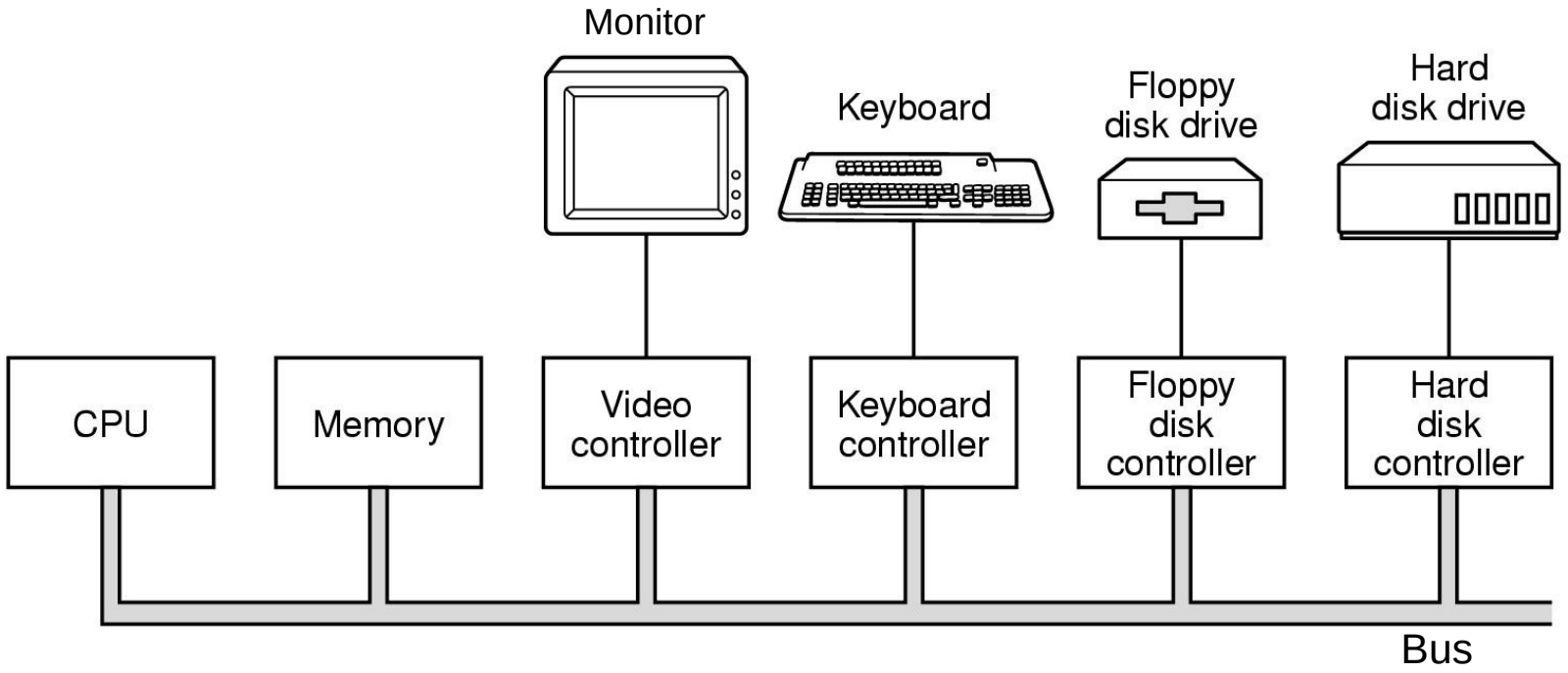
Chapter 5

Input/Output

Principles of I/O hardware
Principles of I/O software
Clocks
Terminals
Disks

Computer hardware: main elements

(*from Introductory chapter*)



Components of a simple (old!) personal computer

Computer I/O architecture

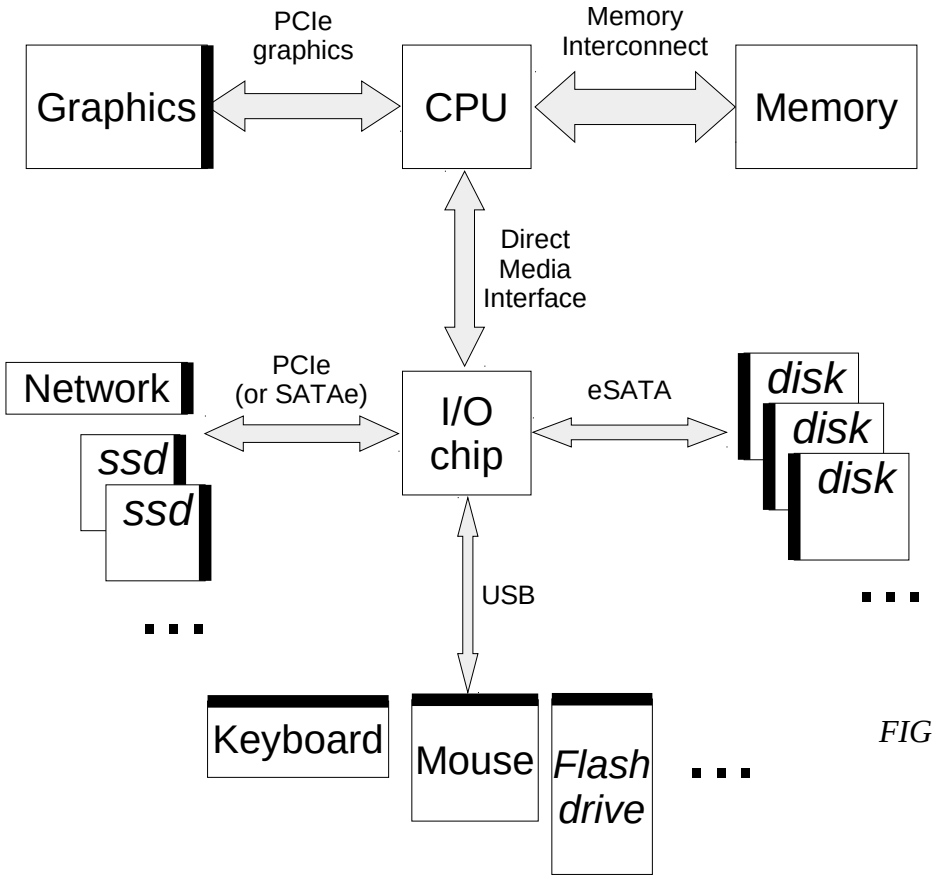


FIG based on Arpaci-Dusseau's OSTEP

Example of modern interconnection of computer components

- bold line represents device controller (sometimes, mainboard has controllers for multiple devices)
- some acronyms' meaning:
  - PCIe: Peripheral Component Interconnect Express
  - SATAe: Serial Advanced Technology Attachment Express
  - eSATA: external Serial Advanced Technology Attachment
  - USB: Universal Serial Bus
  - SSD: Solid State Drive

I/O Devices' speeds

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

Note: most of these figures are dated.
If you have the time, please send the teacher some modern figures, perhaps related to the technologies mentioned in diagram of page 3. Please, accompany each figure with an URL of its origin.
Thank you.

Some typical device, network, and data base rates

I/O Devices

I/O devices have components:

- mechanical component
- electronic component -> device controller (follows device driver's commands) [FIGs ahead]

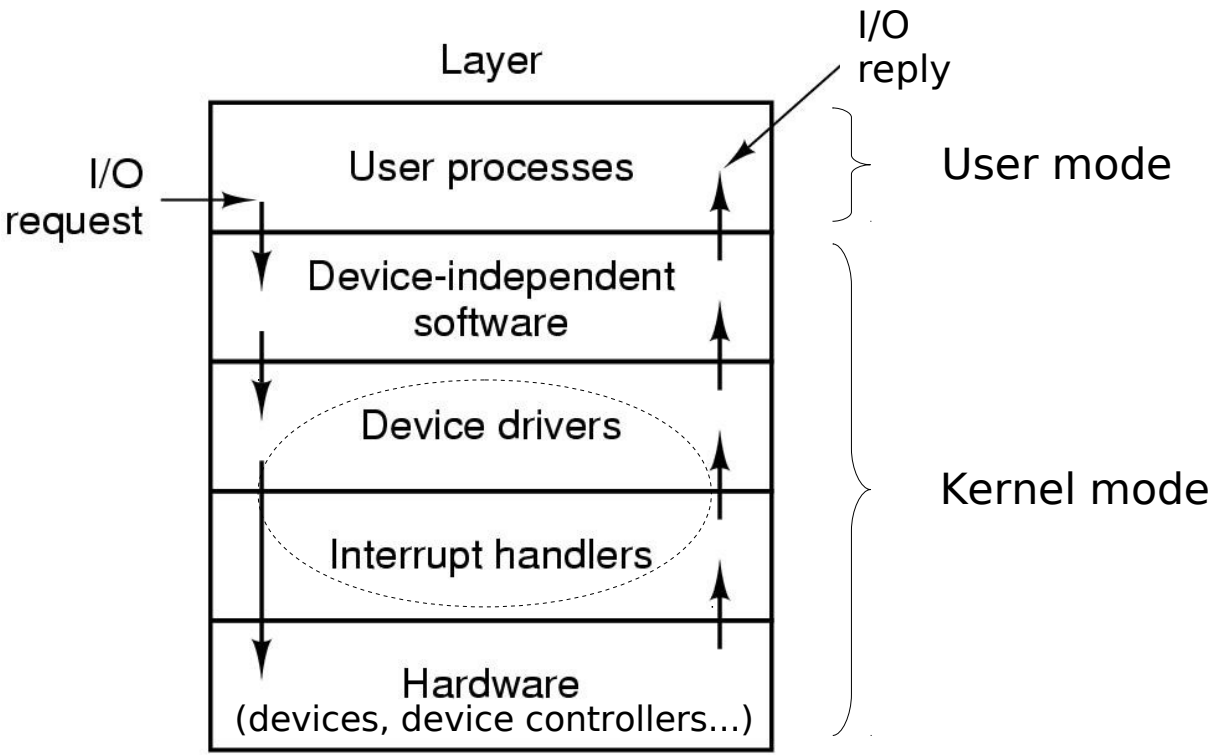
Device controller's tasks:

- convert serial bit stream to block of bytes
- perform error correction as necessary
- make data available for use

Device types:

- *block*: handles addressable chunks of data (random access)
- *character*: handles stream of chars; not seekable
- "other" (clocks...)

I/O Devices' access stack



Device driver programming:

- *polling (programmed I/O)*
- *using interrupts*
- *using Direct Memory Access (DMA)*

I/O Device: programming interface (1)

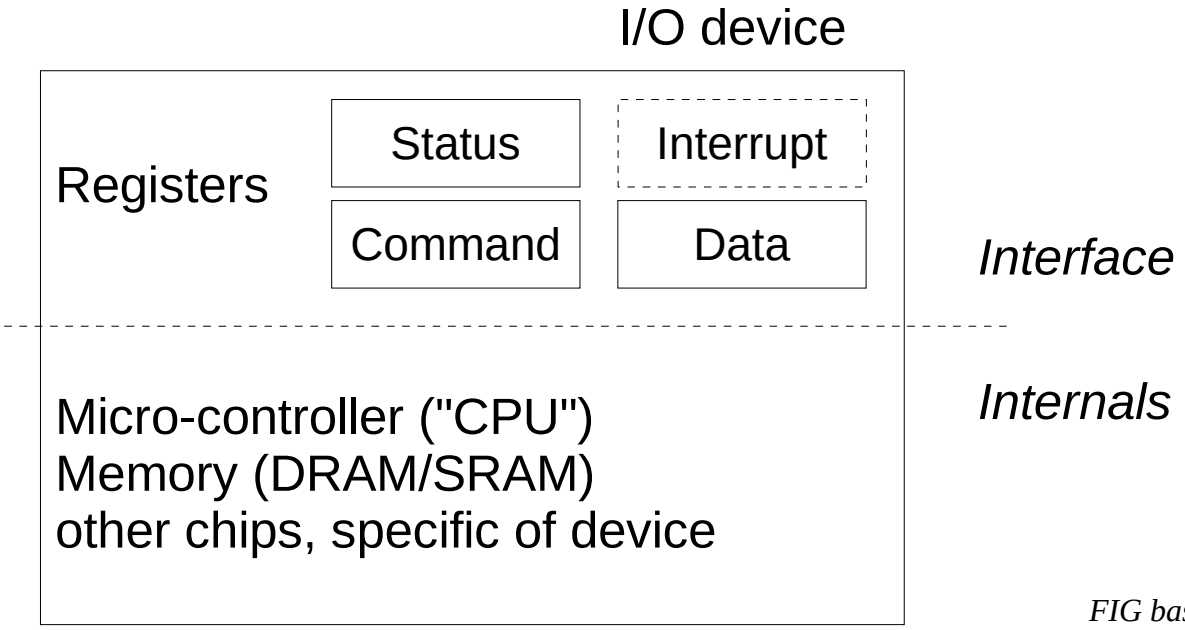


FIG based on Arpaci-Dusseau's OSTEP

I/O device's typical programming access – ***polling***.

Device driver:

1) keeps reading Status, waiting for device to be free
2) writes to Registers Data, then Command
3) keeps reading Status, waiting for device to be free
4) reads eventual answer from Data

I/O Device programming interface (2): *polling*

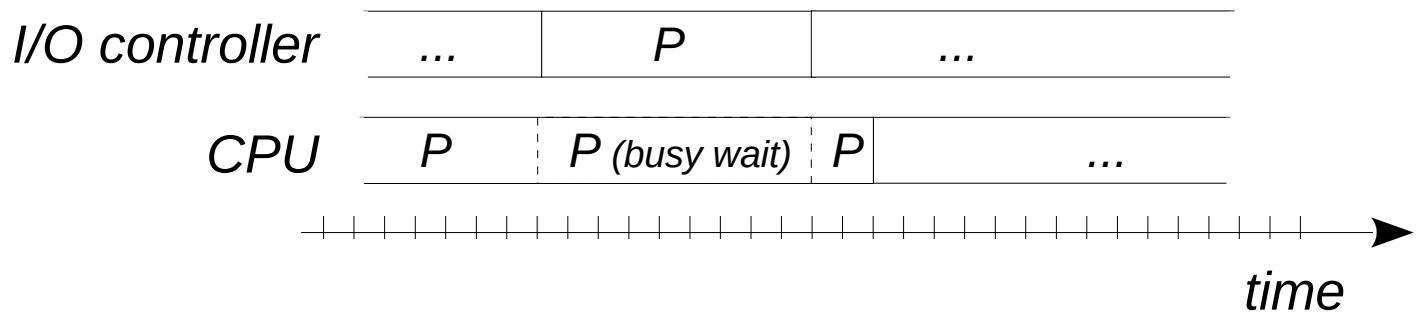
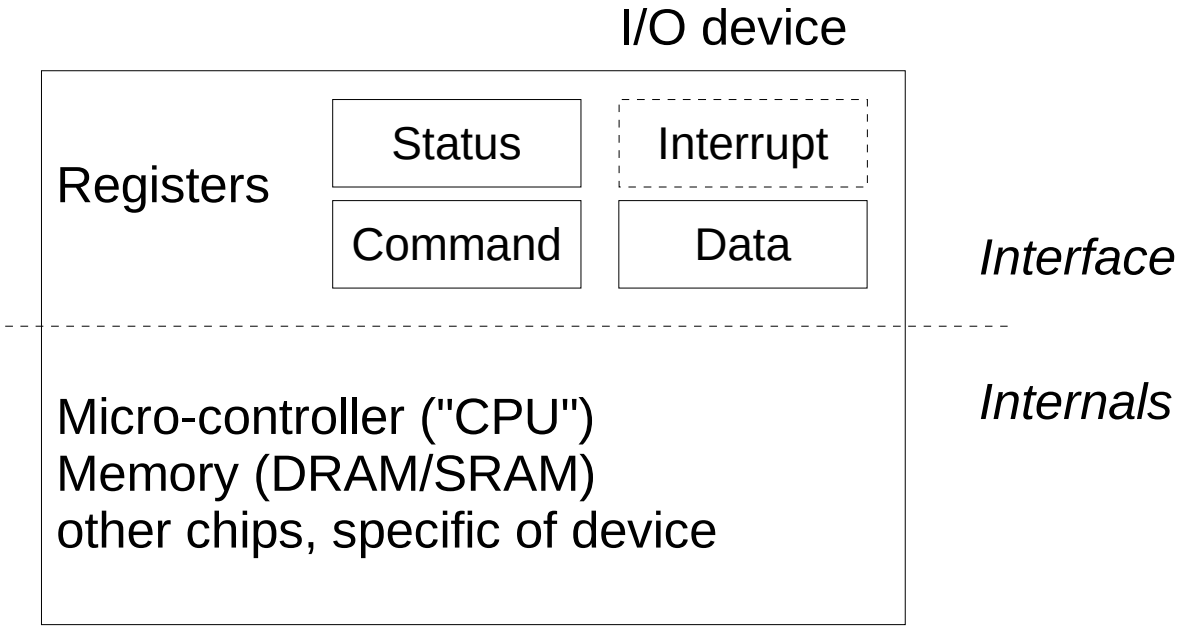


FIG based on Arpaci-Dusseau's OSTEP

Device driver in ***polling*** mode:

```
get_data_from_user (); // put it in kernel buff?
for (all_data) {
  while (device(STATUS) != READY)
    ; // do nothing
  do_IO_operation (some_data);
  }
  return_result_to_user ();
```

I/O Device: programming interface (3)

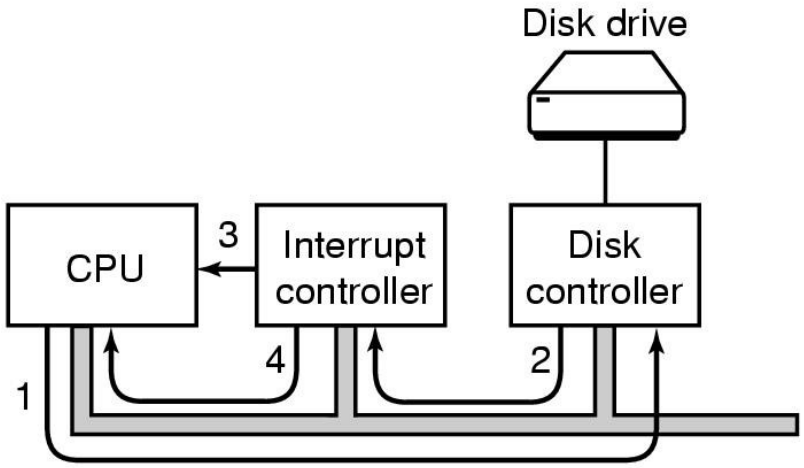


I/O device's *other* typical programming access – ***interrupt mode***. Device driver:

1) keeps reading Status, waiting for device to be free
2) writes to Registers Data, Interrupt, then Command
3) (blocks or does something useful until interrupt is set by controller)
4) reads eventual answer from Data

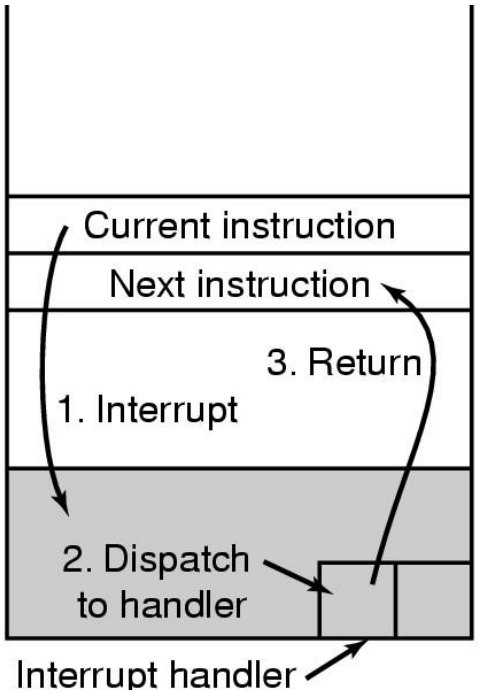
Computer hardware: interrupts

(*from Introductory chapter*)



1) Device driver sets controller's registers
2) Device controller signals Interrupt controller it has finished
3) Interrupt controller signals the CPU a device is ready
4) Interrupt controller identifies I/O device

(a)

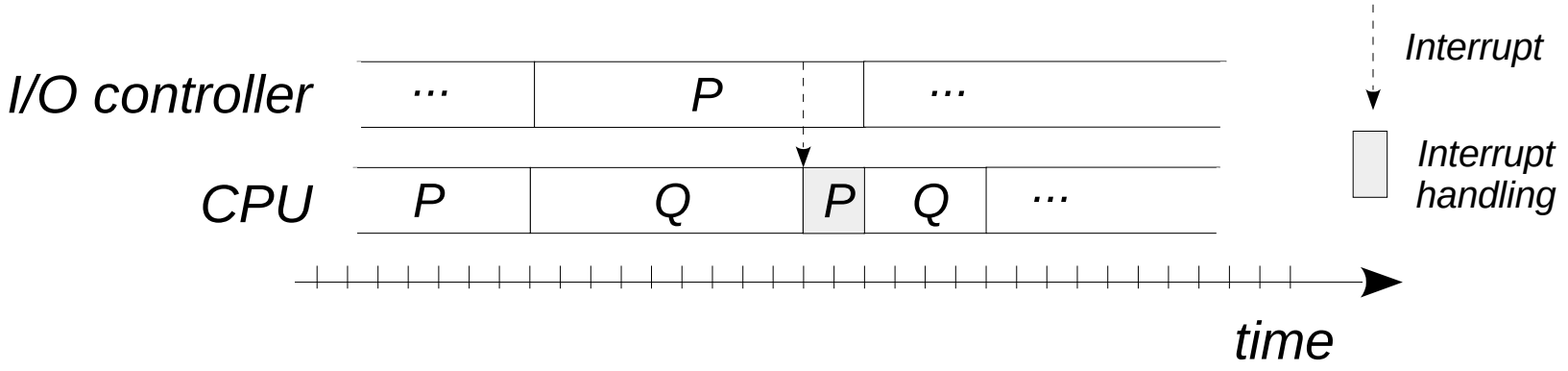


(b)

(a) Steps in starting an I/O device and getting interrupt

(b) How the CPU is interrupted

I/O Device programming interface (4): *using interrupts*

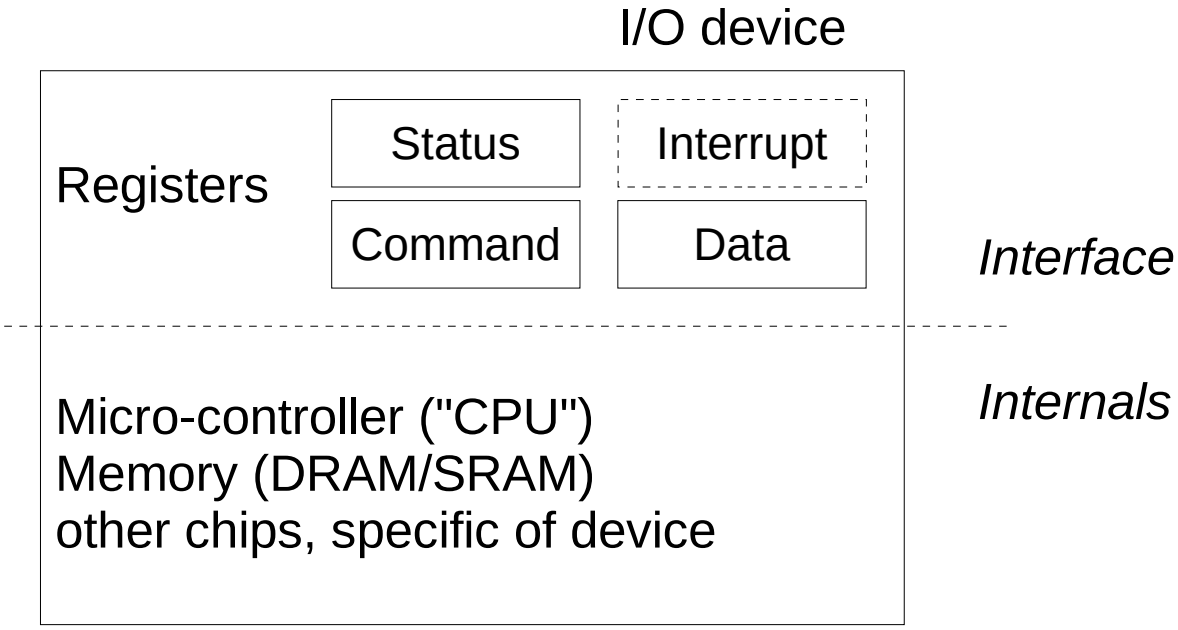


Device driver in ***interrupt*** mode:

```
get_data_from_user (); // kernel buff?
enable_interrupt();
while (device(STATUS) != READY)
  ; // do nothing
do_IO_operation (1st data);
scheduler ();
```

```
// Interrupt service routine:
for (all_data) {
  do_IO_operation (some_data);
  if (no_more_data) {
    return_result_to_user ();
    unblock_user ();
    }
  else break;
  }
  acknowledge_interrupt ();
  return_from_interrupt ();
```

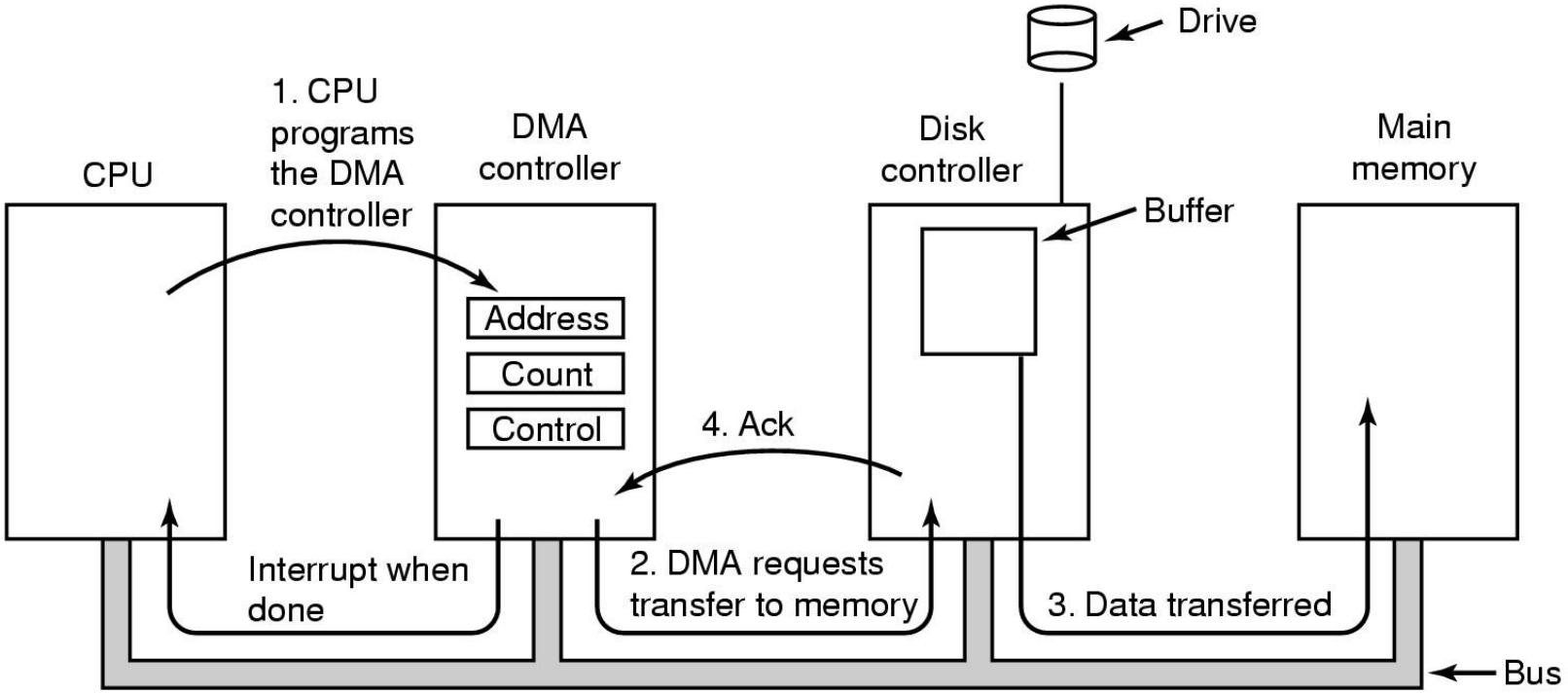
I/O Device: programming interface (5)



I/O device's *other* typical programming access – ***DMA mode***. Device driver:

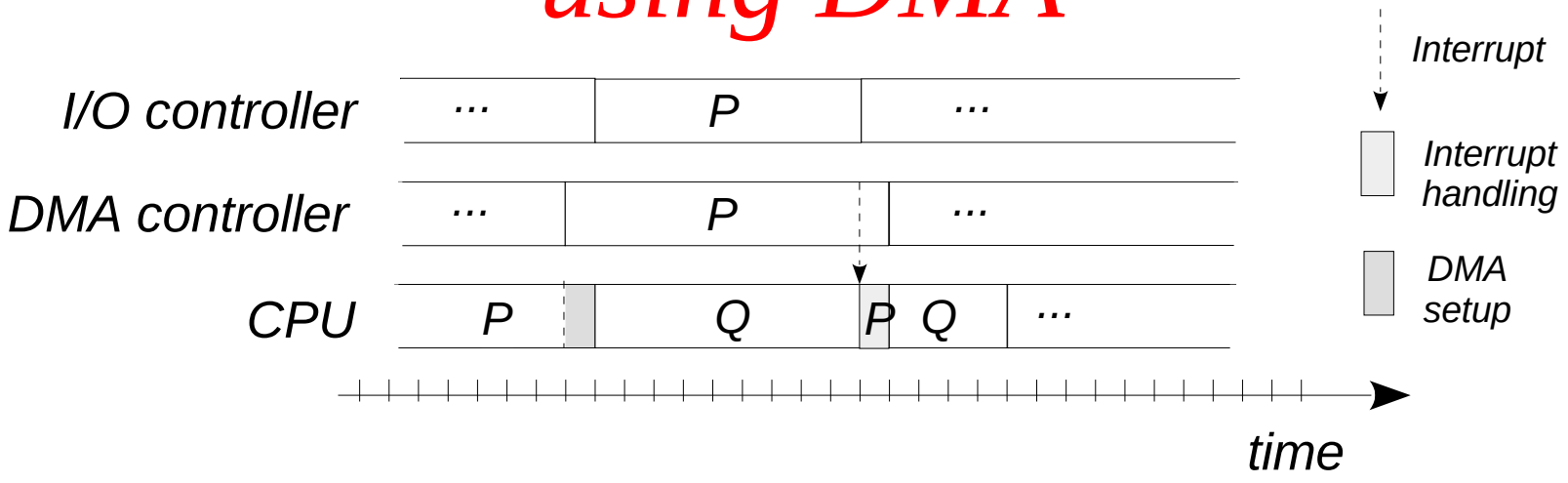
1) sets up Direct Memory Access controller for the I/O access
2) (blocks or does something useful until interrupt is set by DMA controller)
3) eventually processes memory data to return it to user program

Direct Memory Access (DMA)



Operation of a DMA transfer

I/O Device programming interface (6): *using DMA*

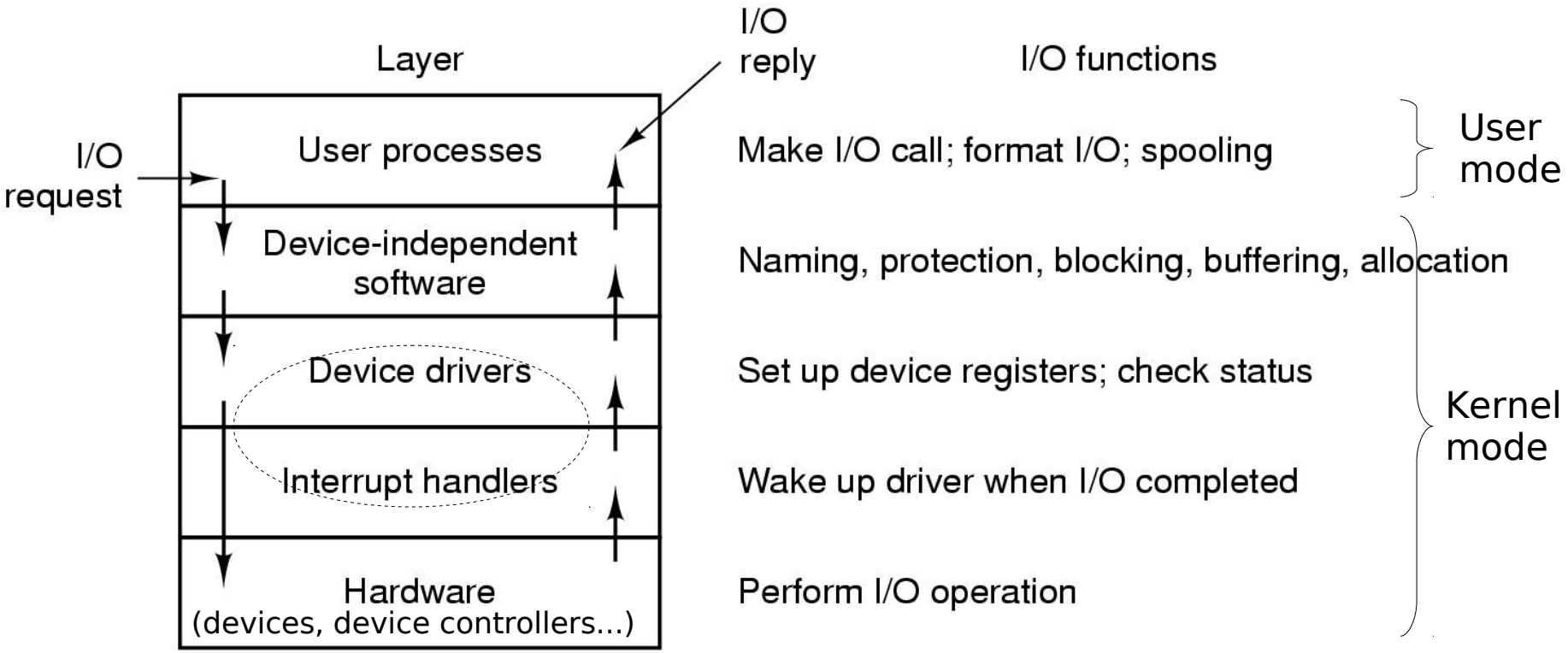


Device driver in ***DMA mode***:

```
get_data_from_user (); // kernel buff?
set_up_DMA();
scheduler ();
```

```
// Interrupt service routine:
return_result_to_user ();
unblock_user ();
acknowledge_interrupt ();
return_from_interrupt ();
```

Structure of I/O Software



Goals of I/O Software

Device independence

- programs access any I/O device (hard drive, flash disk, network, ...) the same way

Uniform naming

- names of files or devices are strings or integers, independent of physical media or machine

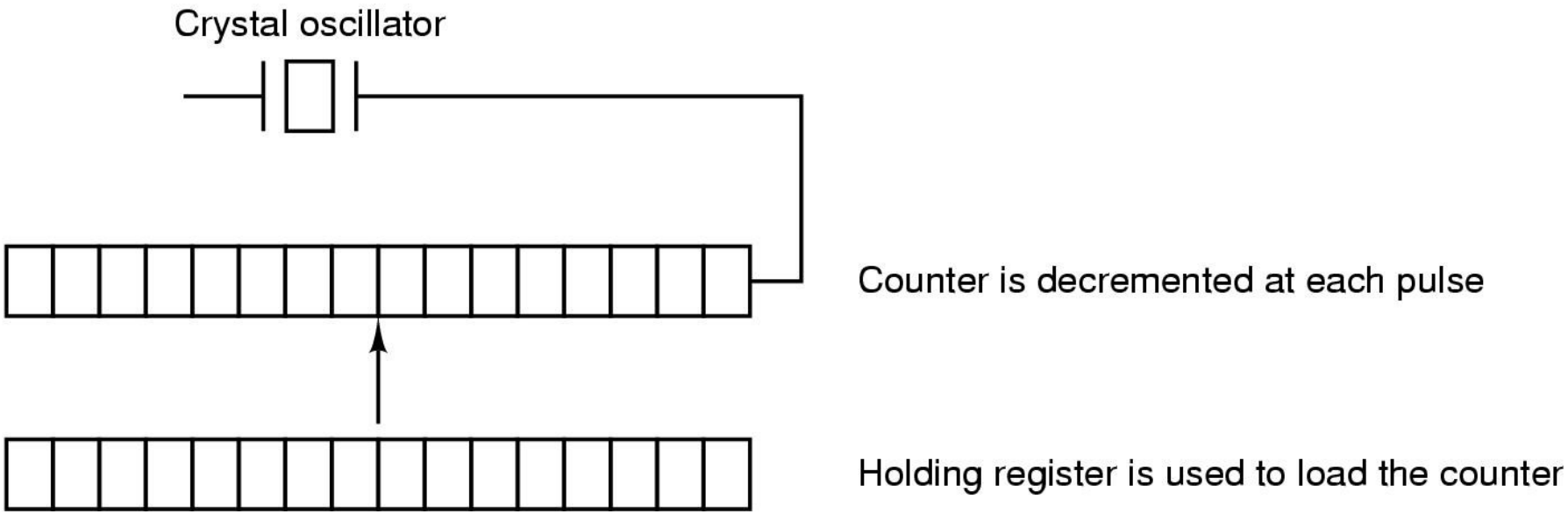
Error handling

- handle as close to the hardware as possible

Buffering

- intermediary data buffers are not desirable (but many times necessary: kernel/user address spaces)

Clocks: Hardware



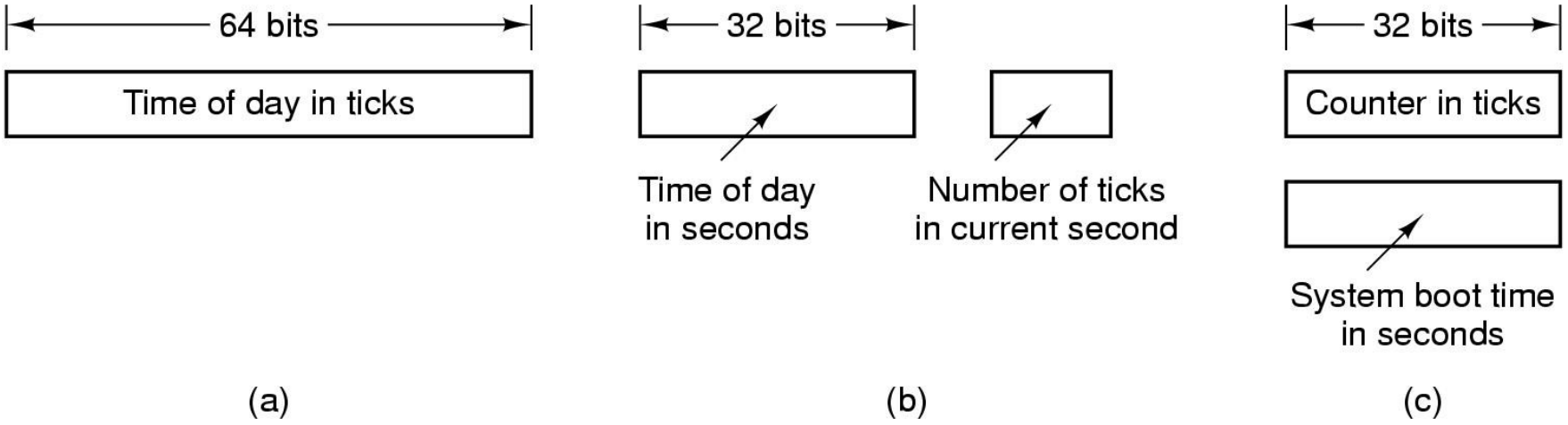
A programmable clock

Clock Software (1)

Chores:

- maintain *time of day*
- call processes' scheduler
- account for processes' CPU usage
- service alarms' calls (user's & system's)
- do profiling, monitoring, statistics gathering

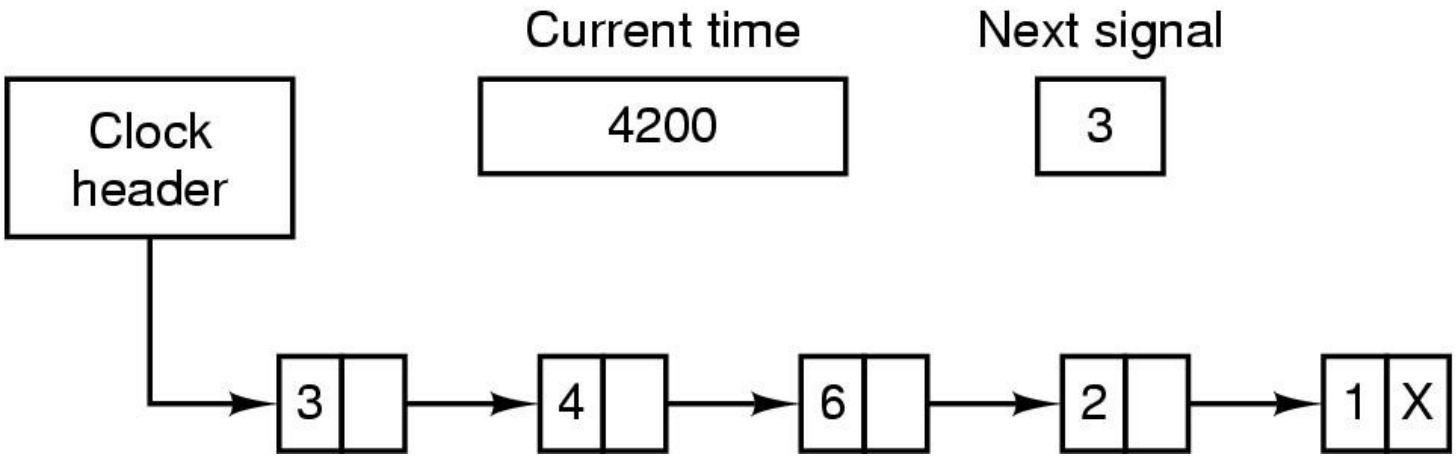
Clock Software (2):
maintain *time of day*



Three ways to maintain the time of day.

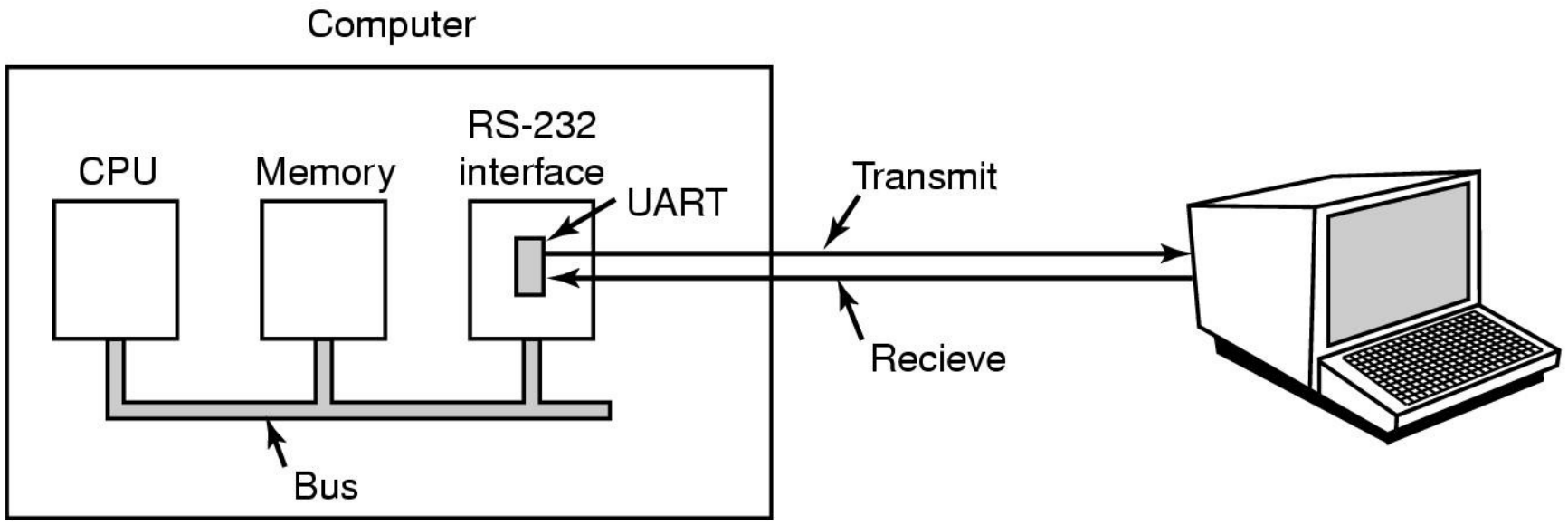
Note (100 Hz clock): $2^{32}/(100*60*60*24*365)$ ~ 1,4 years

Clock Software (3):
service alarms' requests



Simulating multiple timers with a single clock

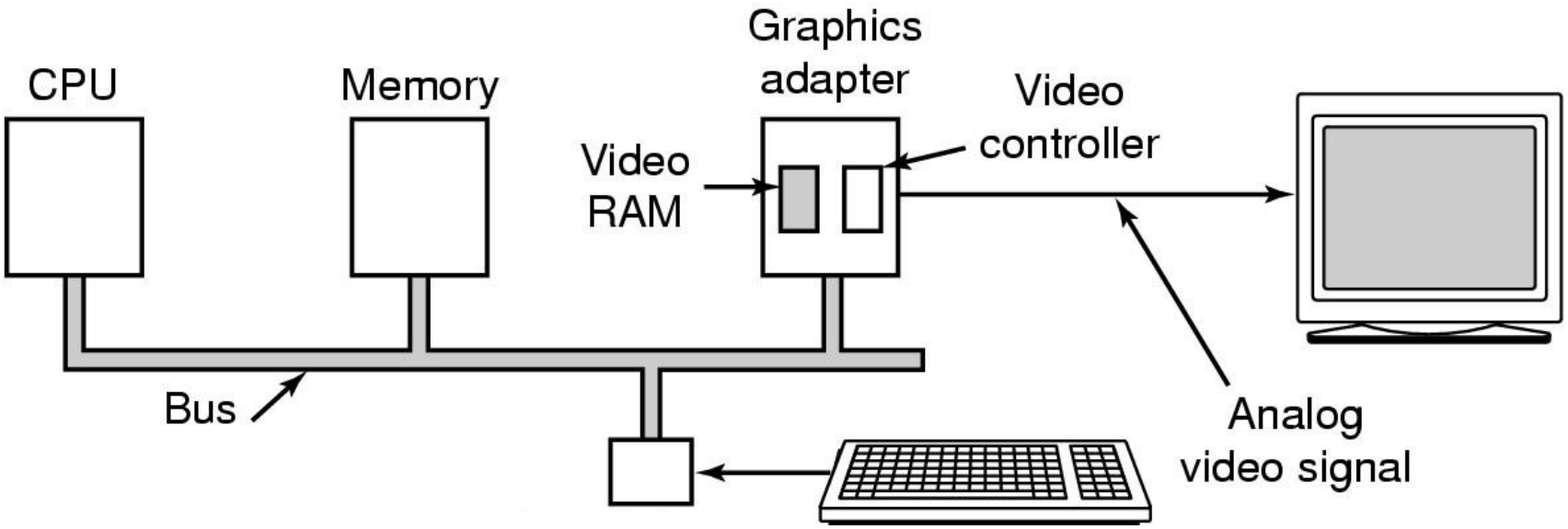
Character Oriented Terminals
RS-232 Terminal Hardware



An RS-232 terminal communicates with computer 1 bit at a time

Computer and terminal are completely independent

Display Hardware (1)



Memory-mapped displays:
driver writes directly into display's video RAM

Input Software

Keyboard controller delivers a (scan) code

- pressing & releasing of key

Keyboard driver delivers a number

- driver converts to characters & control data
- uses a ASCII table & control table
- combination of keys (e.g. CTRL+SHIFT+A)

Exceptions, adaptations needed for other languages

- many OS provide for loadable keymaps or code pages

Input Software (2)

Keyboard driver: modes of operation

	character oriented	*line oriented*
Unix-lingo:	raw	cooked
POSIX-lingo:	non-canonical	canonical

Input Software (3)

Character	POSIX name	Comment
CTRL-H	ERASE	Backspace one character
CTRL-U	KILL	Erase entire line being typed
CTRL-V	LNEXT	Interpret next character literally
CTRL-S	STOP	Stop output
CTRL-Q	START	Start output
DEL	INTR	Interrupt process (SIGINT)
CTRL-\	QUIT	Force core dump (SIGQUIT)
CTRL-D	EOF	End of file
CTRL-M	CR	Carriage return (unchangeable)
CTRL-J	NL	Linefeed (unchangeable)

Characters handled specially in canonical mode

Output Software

Escape sequence	Meaning
ESC [*n* A	Move up *n* lines
ESC [*n* B	Move down *n* lines
ESC [*n* C	Move right *n* spaces
ESC [*n* D	Move left *n* spaces
ESC [*m* ; *n* H	Move cursor to (*m*,*n*)
ESC [*s* J	Clear screen from cursor (0 to end, 1 from start, 2 all)
ESC [*s* K	Clear line from cursor (0 to end, 1 from start, 2 all)
ESC [*n* L	Insert *n* lines at cursor
ESC [*n* M	Delete *n* lines at cursor
ESC [*n* P	Delete *n* chars at cursor
ESC [*n* @	Insert *n* chars at cursor
ESC [*n* m	Enable rendition *n* (0=normal, 4=bold, 5=blinking, 7=reverse)
ESC M	Scroll the screen backward if the cursor is on the top line

Some ANSI escape sequences accepted by terminal driver on output

- ESC is ASCII character (0x1B)
- *n*, *m* and *s* are optional numeric parameters

Terminal Control (1)

```
struct termios {
  tcflag_t c_iflag;   /* input modes */
  tcflag_t c_oflag;   /* output modes */
  tcflag_t c_cflag;   /* control modes */
  tcflag_t c_lflag;   /* local modes */
  speed_t c_ispeed;   /* input speed */
  speed_t c_ospeed;   /* output speed */
  cc_t c_cc[NCCS];    /* control characters */
};
```

c_lflag	ICANON	Enable canonical mode
	ISIG	When any of the characters INTR (CTRL-C), QUIT... are received, generate the corresponding signal.
	ECHOE	If ICANON is also set, the ERASE (CTR-H) character erases the preceding input character...
	ECHONL	If ICANON is also set, echo the NL character even if ECHO is not set.
	TOSTOP	Send the SIGTTOU signal to the process group of a background process which tries to write to its controlling terminal.
	...	...

Controlling the character terminal: the *termios* structure and some of its flags

Terminal Control (2)

```
struct termios tms, tms_ini;
char c;

tcgetattr(STDIN_FILENO, &tms_ini); // read console's config
tms = tms_ini;
tms.c_lflag &= ~ECHO;                  // inhibit char echoing
tcsetattr(STDIN_FILENO, TCSANOW, &tms);     // set new config

do { read(STDIN_FILENO, &c, 1); }
       while (c != '\n');              // wait for [ENTER]

tcsetattr(STDIN_FILENO, TCSANOW, &tms_ini); // reset config
```

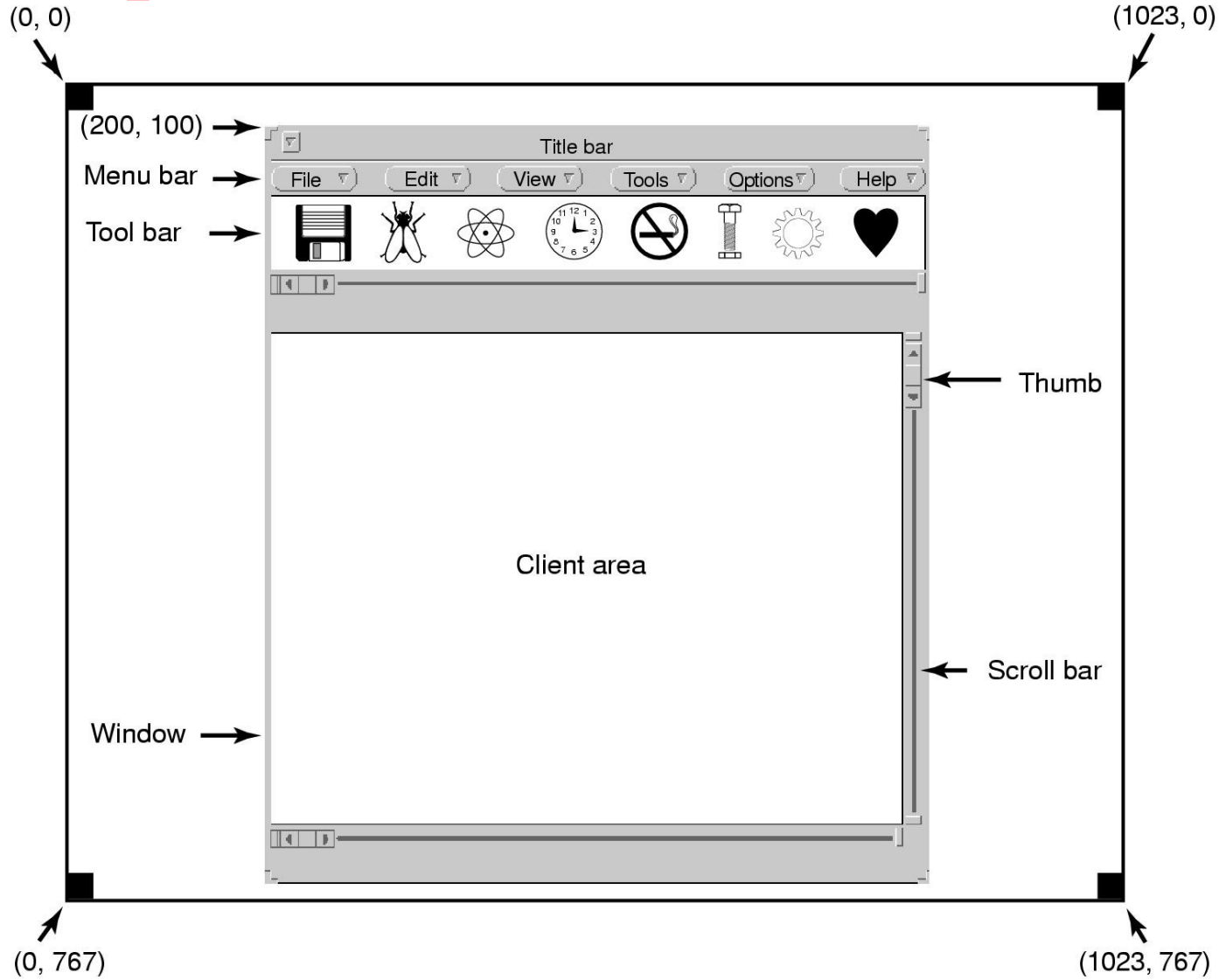
Code segment for inhibiting the echoing of typed characters

Graphical User Interfaces:
Enter the Dragon!...

Original Apple's MacIntosh
(1984)

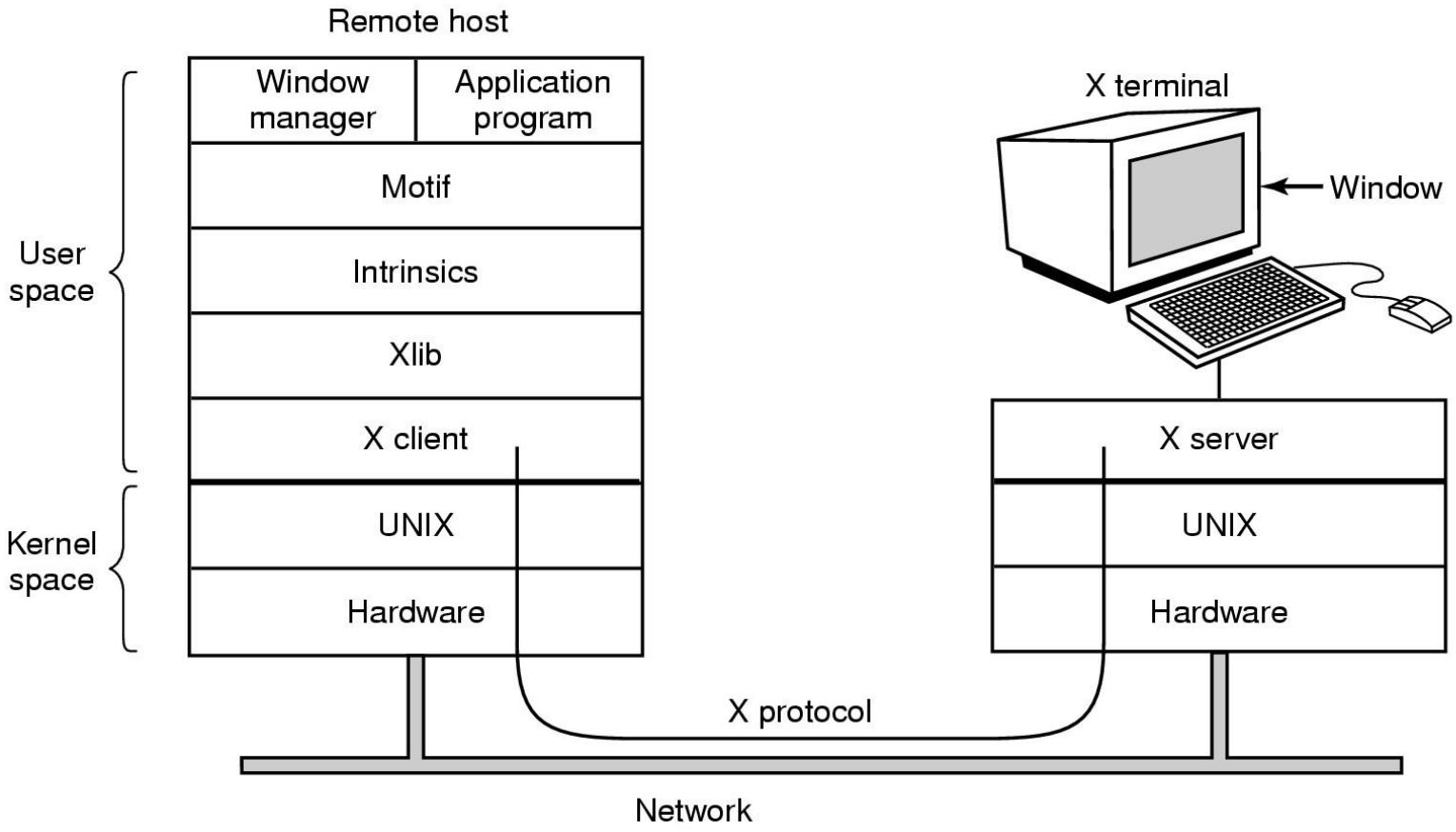
in http://www.aresluna.org/attached/computerhistory/articles/macintoshbytepreview

Graphical User Interfaces (1)



Sample window located at (200,100) on XGA display

Network Terminals
X Windows



Clients and servers in the M.I.T.'s X Window System

Disks: Hard Disk Drives (1)

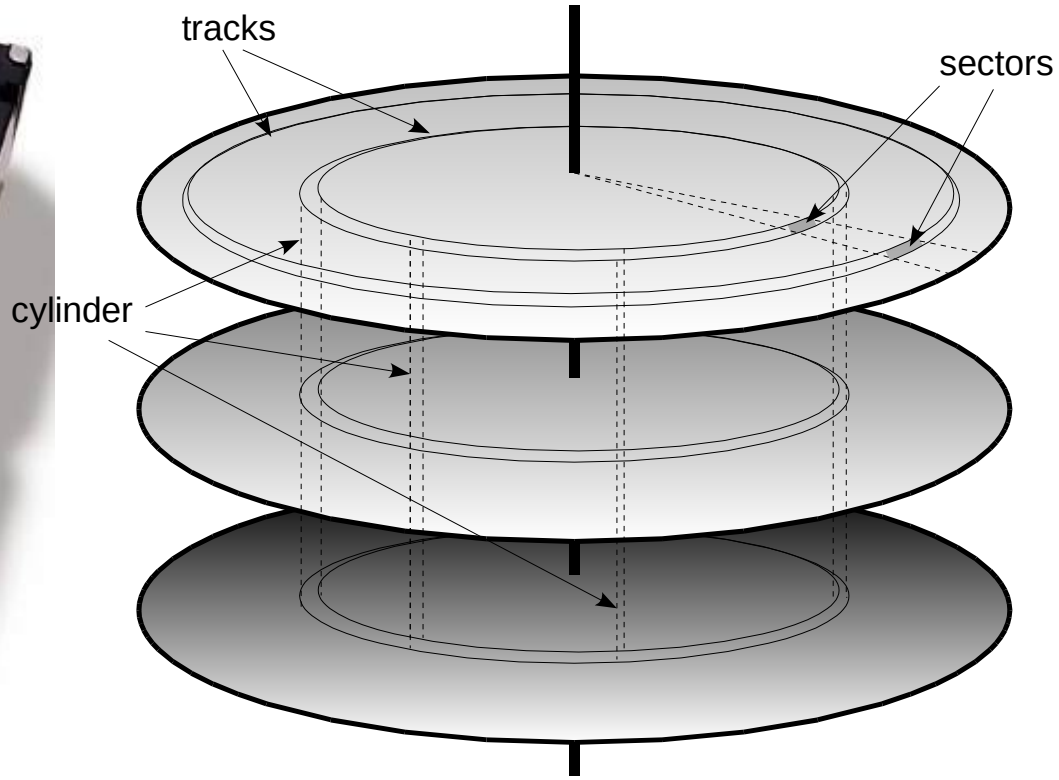
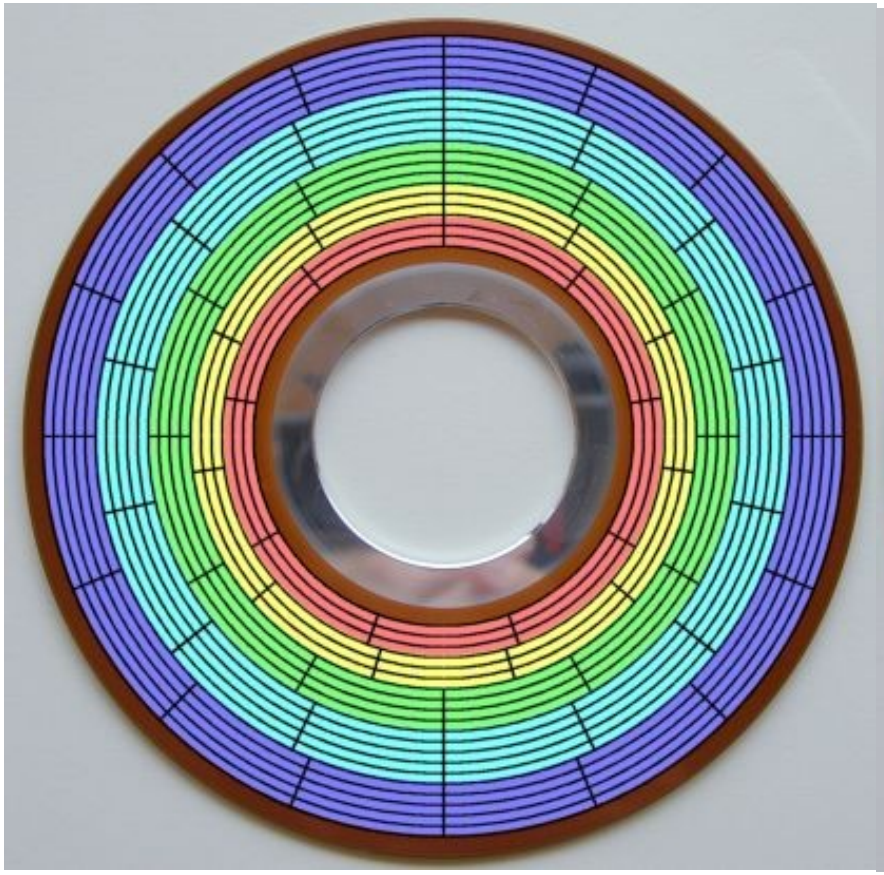
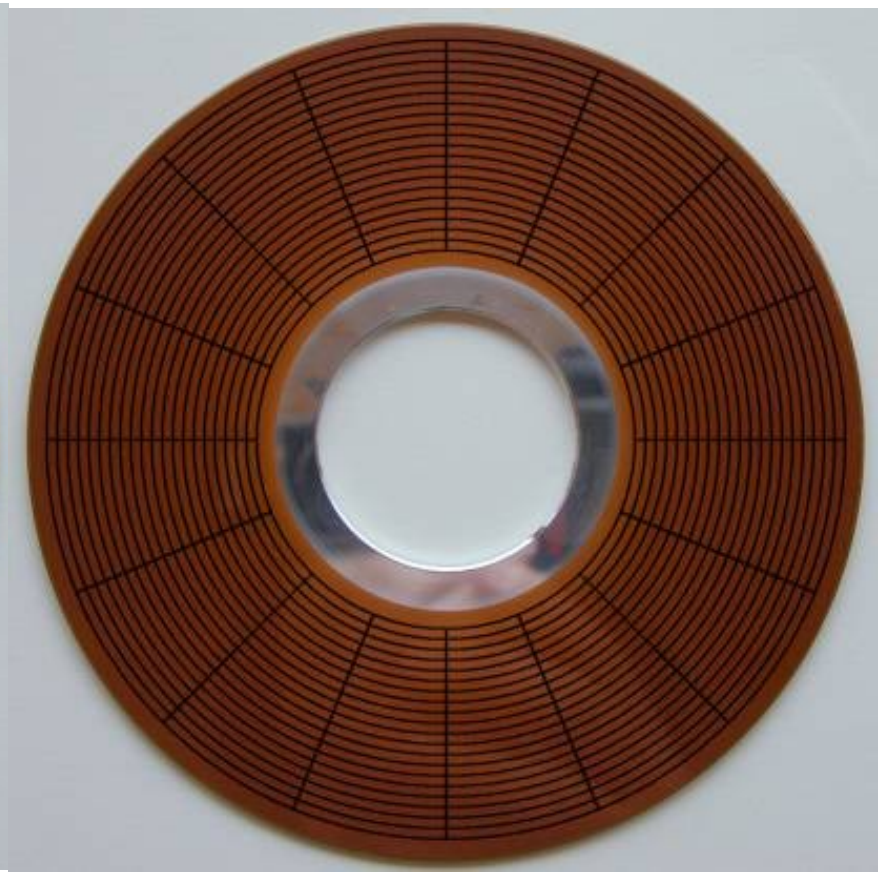


Photo of Hard Disk Drive (HDD) and schematics of disk geometry

Hard Disks geometry (2)



Physical geometry of a disk with zoned bit recording



A possible virtual geometry for this disk

needed for not "breaking" old software

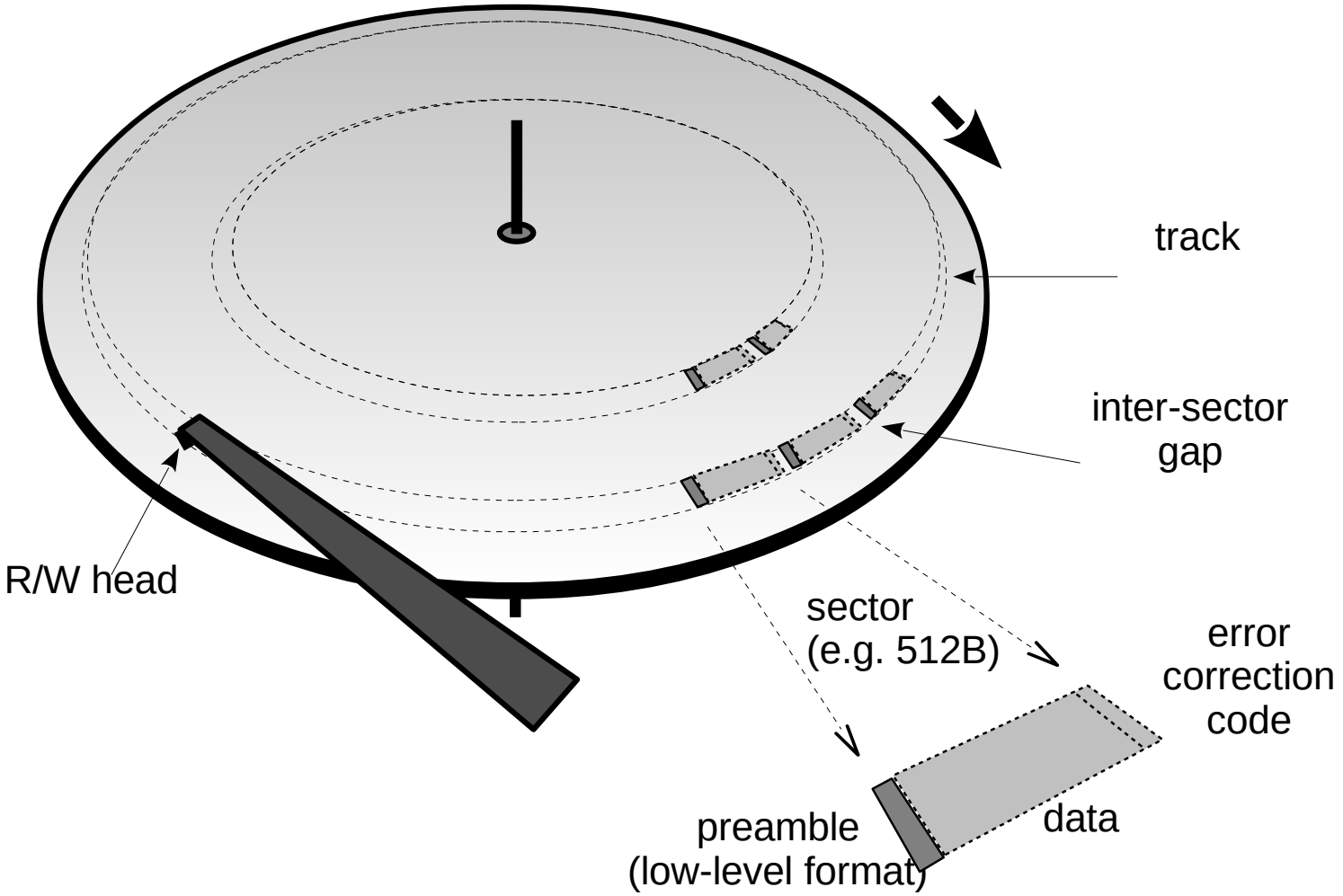
Hard Disks: access geometry

Disk structure: low-level

Modern disks completely hide inner physical geometry: access is through LBA (Logical Block Addressing)

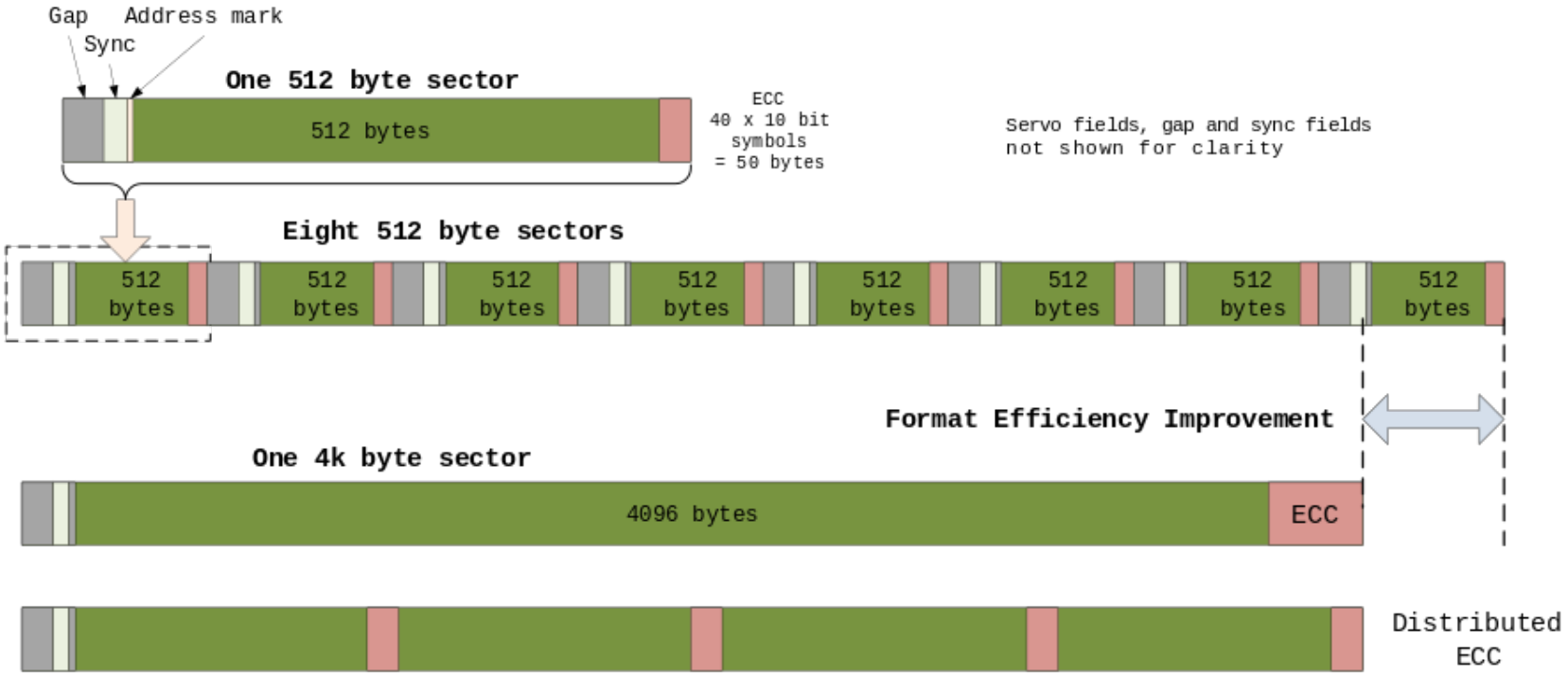
Logical Block Addressing is a linear addressing scheme: one number is used to address a chunk of data, called block (or sector).

Hard Disk Formatting (1)



Disk sector structure and low-level formatting

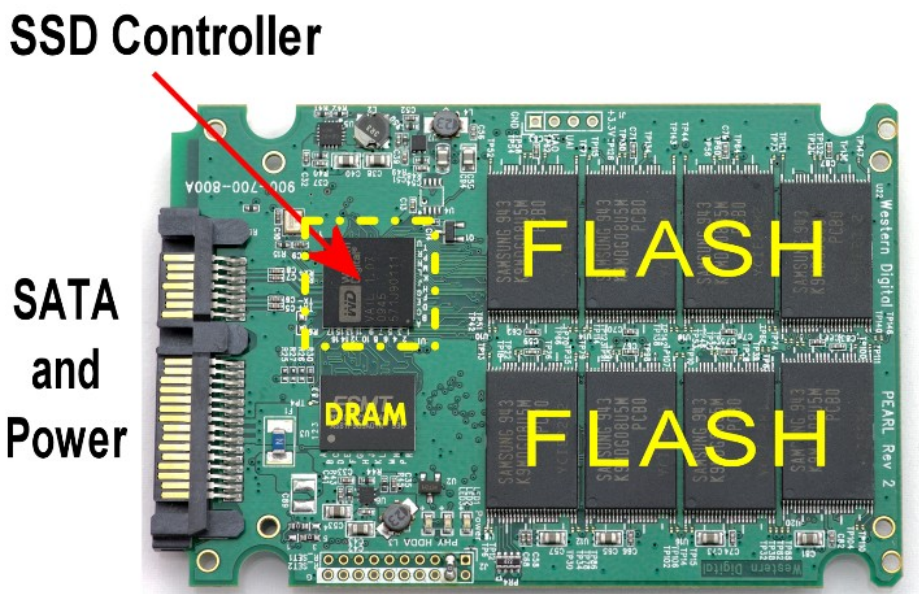
Hard Disk Formatting (2)



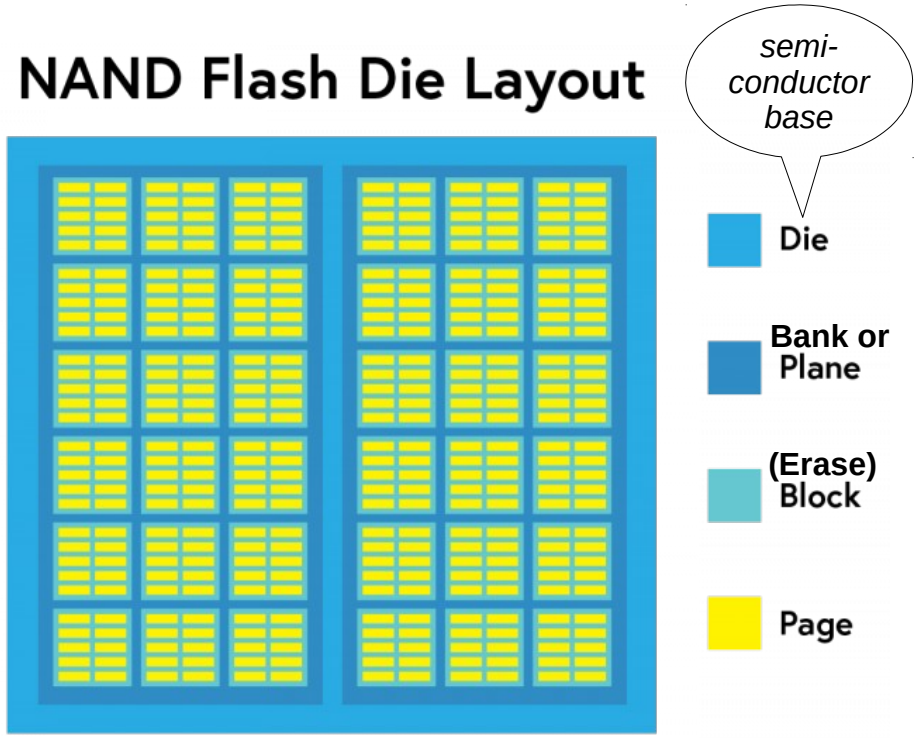
in Wikipedia (Advanced_Format)

Disk sector sizes: traditional (512 B) vs modern (4096 B)

Disk: Solid State Drive (1)



in www.storagereview.com



in Wikipedia (Advanced_Format)

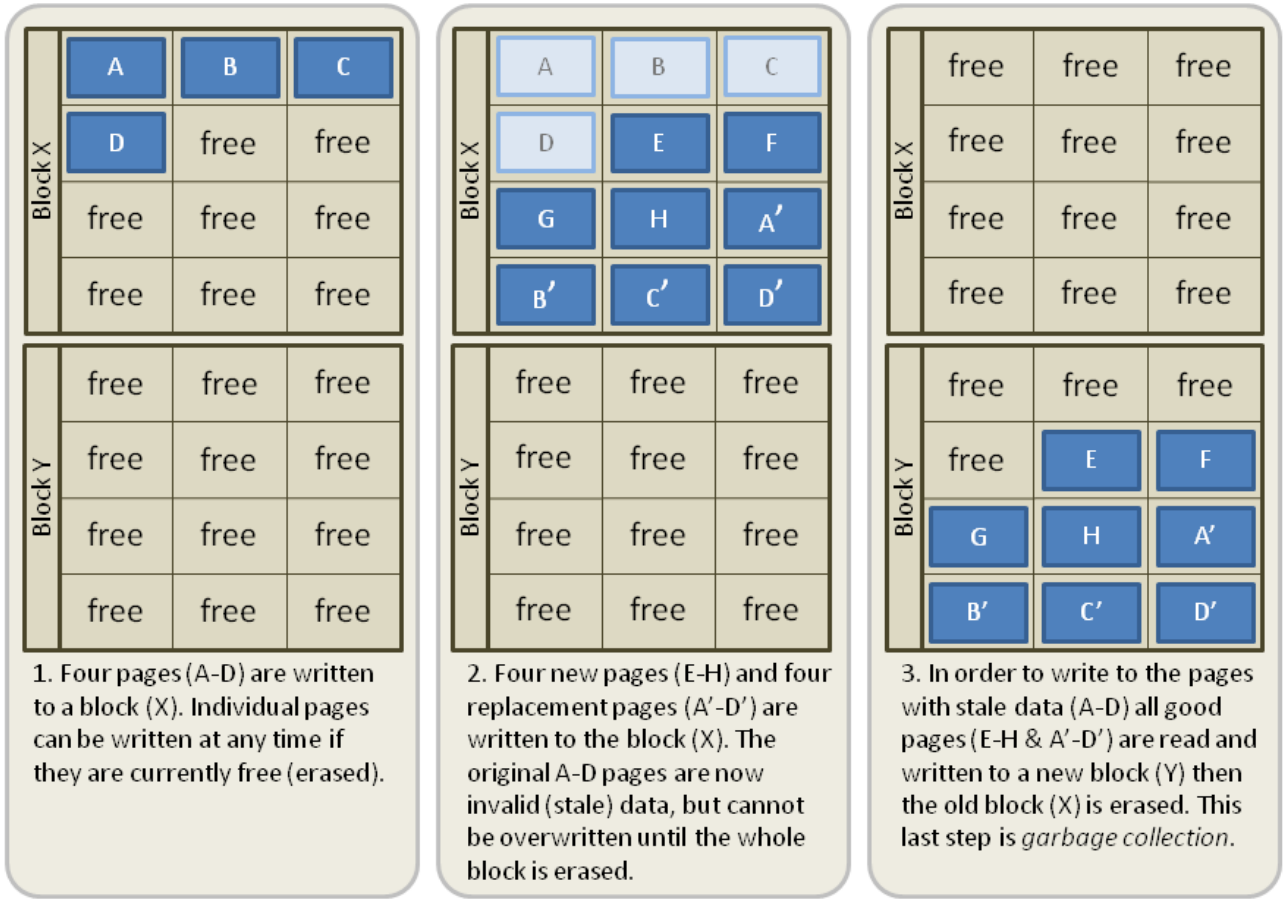
flash block: 128 – 256 pages
flash page: 512 – 4,096 bytes

Photo of Solid State Disk and schematics of Flash memory

Disk: Solid State Drive (2)

Accessing Flash storage:

- read a flash-page:
  - ~ 10 us
- write a flash-page (*if it is erased*):
  - ~ 100 us
- erase a flash-**block**:
  - ~ 1000 us



in Wikipedia (Write_amplification)

Solid State Disks (Flash-based): re-writing problem!

Disks technology: brief comparison

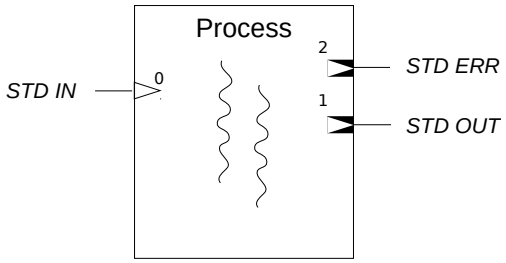
Vendor	Seagate	Seagate
Year	2020	2019
Model	ST500DM009	ZA500CM10003
Technology	Hard Disk Drive	Solid State Drive (3D TLC)
Capacity	500 GB	500 GB
Maximum (*) Transfer Data Rate	210 MB/s	550 MB/s
Operating power	5,3 W	2,6 W
Connection	SATA 6 Gb/s	SATA 6 Gb/s

(*) probably, sequential

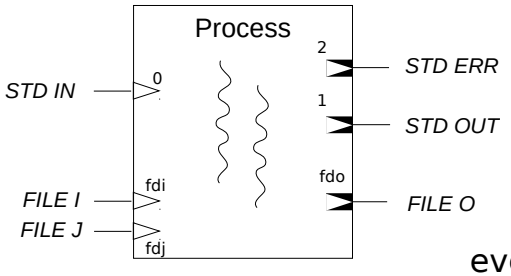
Some disk parameters for 2 current disks of the same vendor

Annex: Input/Output redirection programming

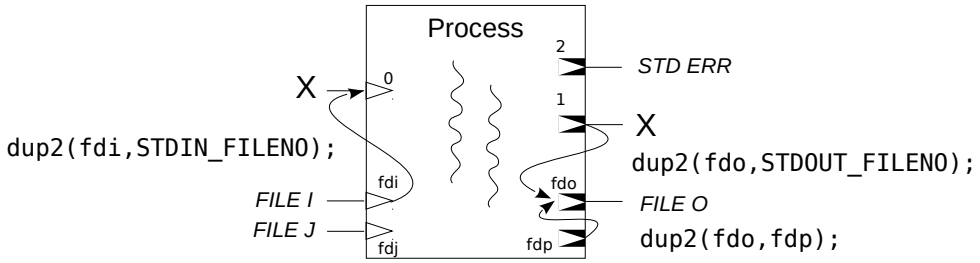
initially:



later:



even later, after 3 dups:



see also in Moodle's SOPE area:

- *"POSIX dups: duplicating file descriptors"*

By redirection, a program can seamlessly change access to different I/O devices than the ones initially opened.