Embedded Operating Systems

Prof. Otávio Gomes
otavio.gomes@unifei.edu.br
in /otavio-gomes

I/O System

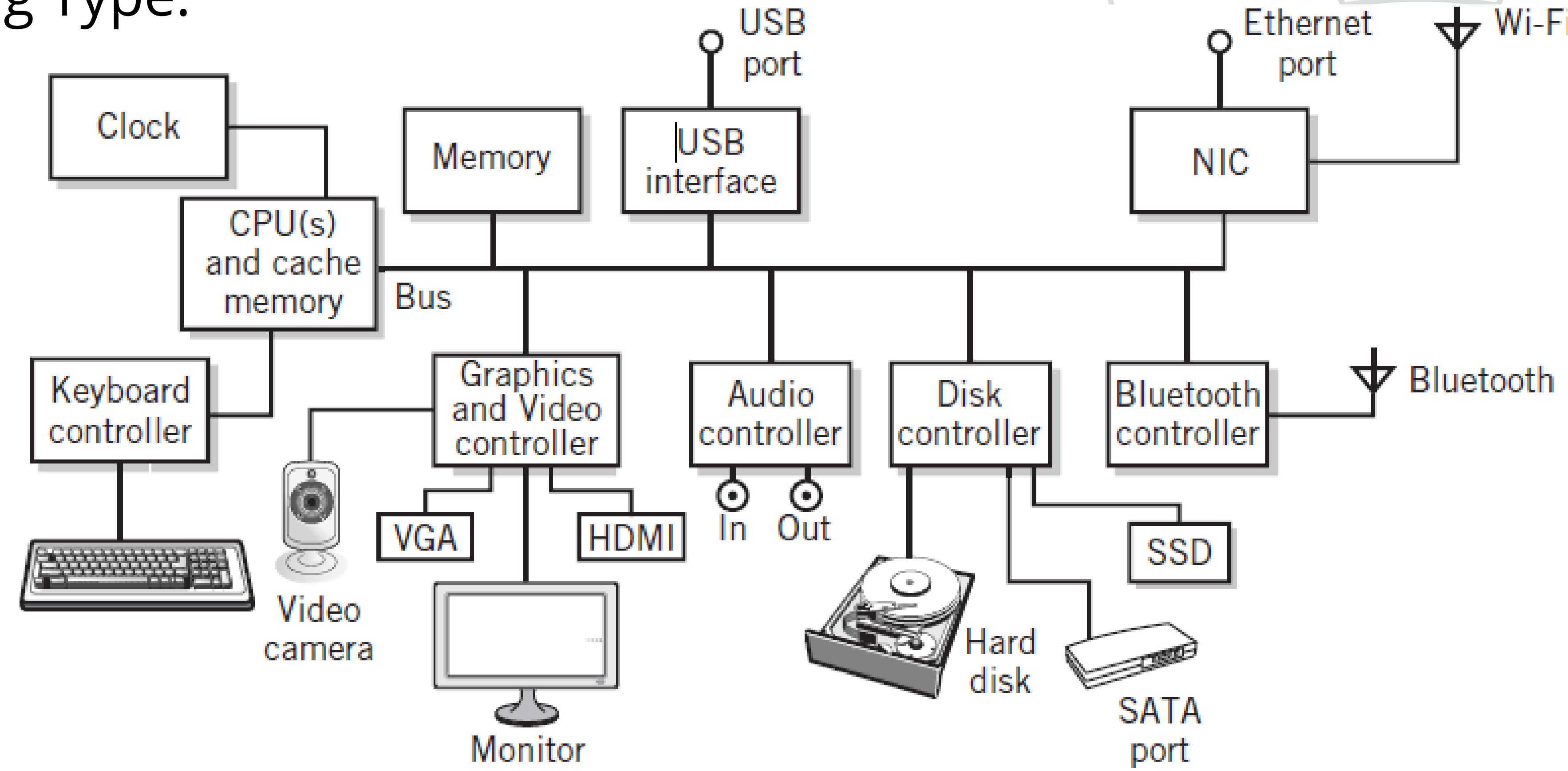
Overview

- The OS controls all I/O devices, having commands for:
  - Run instructions (read, write, etc.);
  - Intercepting interruptions;
  - Handling errors.

- A single interface (**API – Application Program Interface**) for all devices due to:
  - Differences in device construction;
  - Device processing time;
  - Device access time.

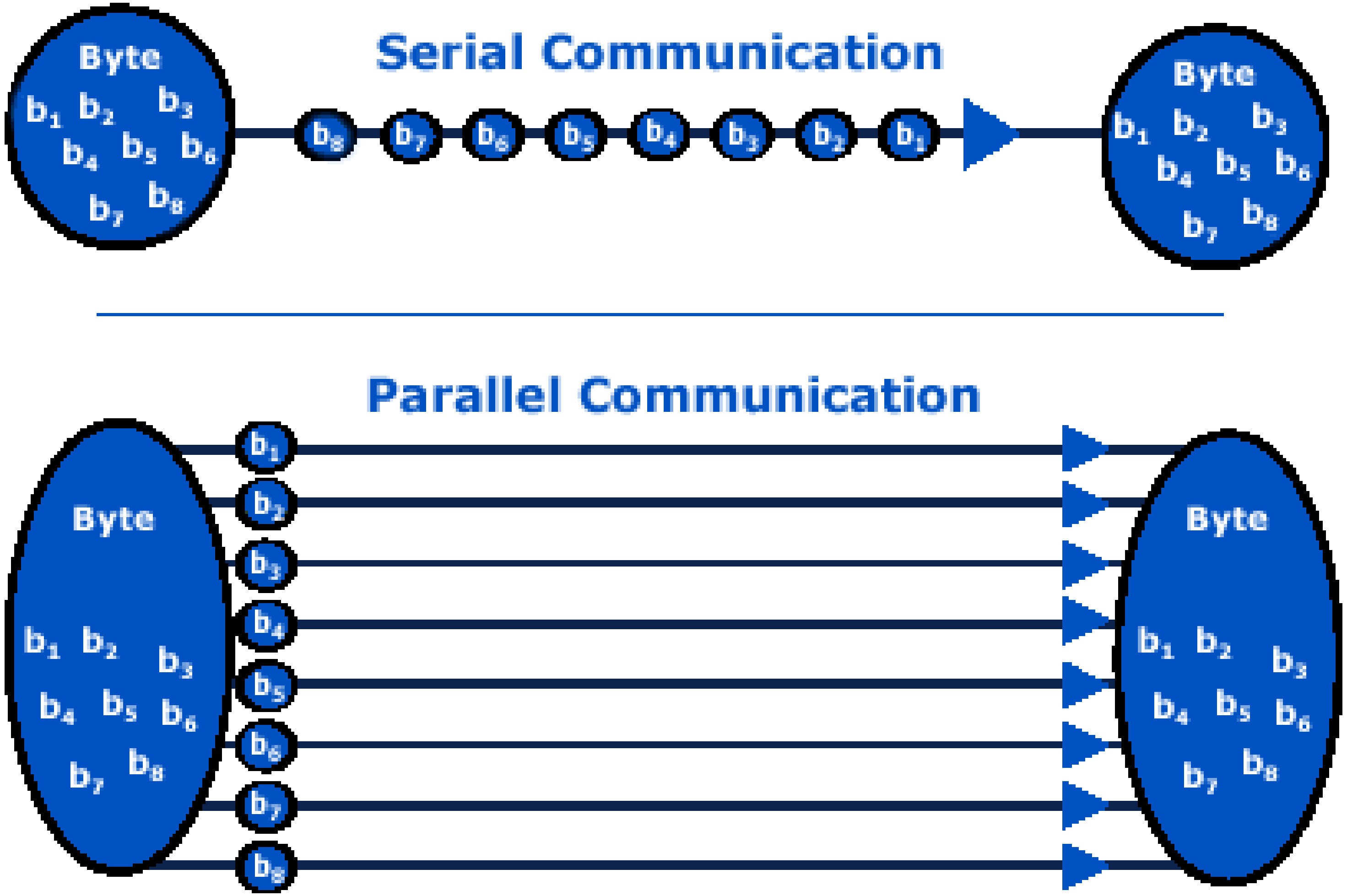
They can be classified according to the following types:

- Connection Type;
- Data Transfer Type;
- Connection Sharing Type.



1) Connection Type

- The nature of the connection between the I/O module and peripheral
- **Serial** versus **Parallel.**



1) Connection Type

- The nature of the connection between the I/O module and peripheral
- **Serial** versus **Parallel**

a) Serial – a single connection line.

- Advantages: cheaper than parallel
- Disadvantages: slower than parallel (context / technology).

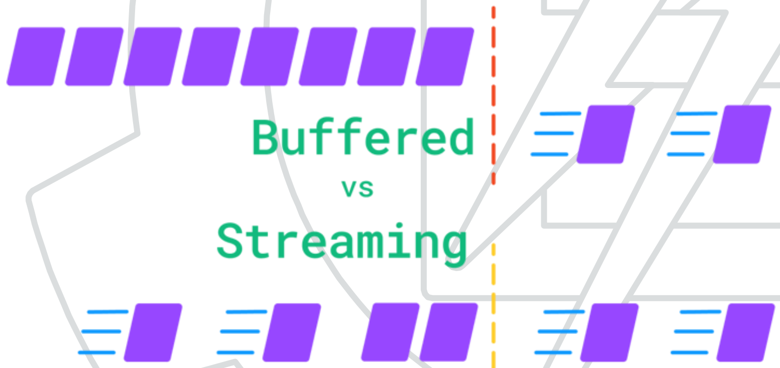
b) Parallel – multiple connection lines.

- Advantages: faster than serial (context / technology).
- Disadvantages: more expensive than serial (buses).

2) Data Transfer Type

a) **Block** devices:

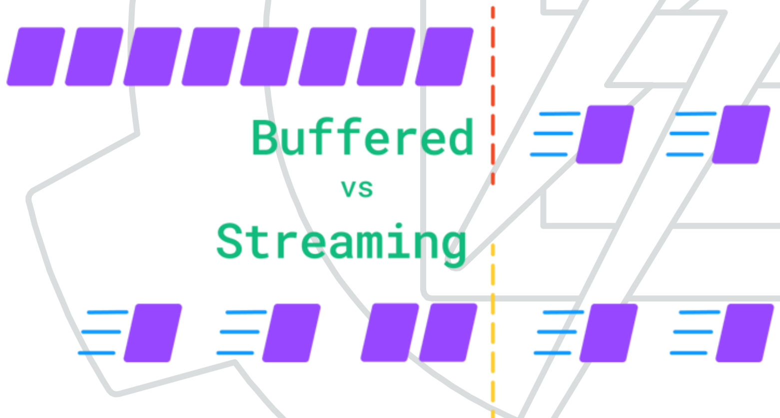
- Fixed-size blocks, each with its address.
- Size from 128 to 1024 bytes.
- Transfer with one or more blocks.
- Locality reference – Optimization of reading.
- I/O consumes time.
- Examples: HDD, CD-ROM, USB Drive, etc.



2) Data Transfer Type

b) **Character** devices:

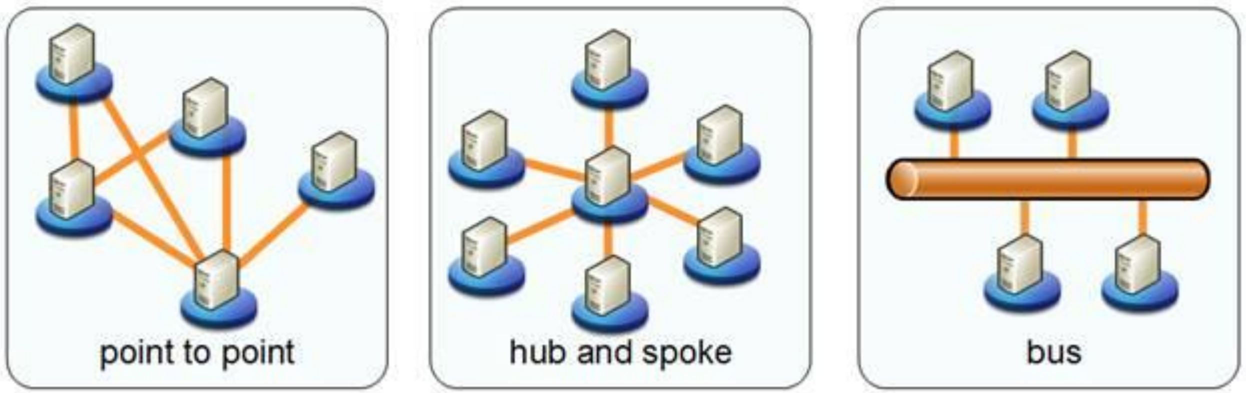
- Access a stream of characters.
- Do not consider blocks and are not addressable.
- Do not have random access (seek operation).
- Example: network interfaces, mouse, etc.



3) Connection Sharing Type

a) **Point-to-point**

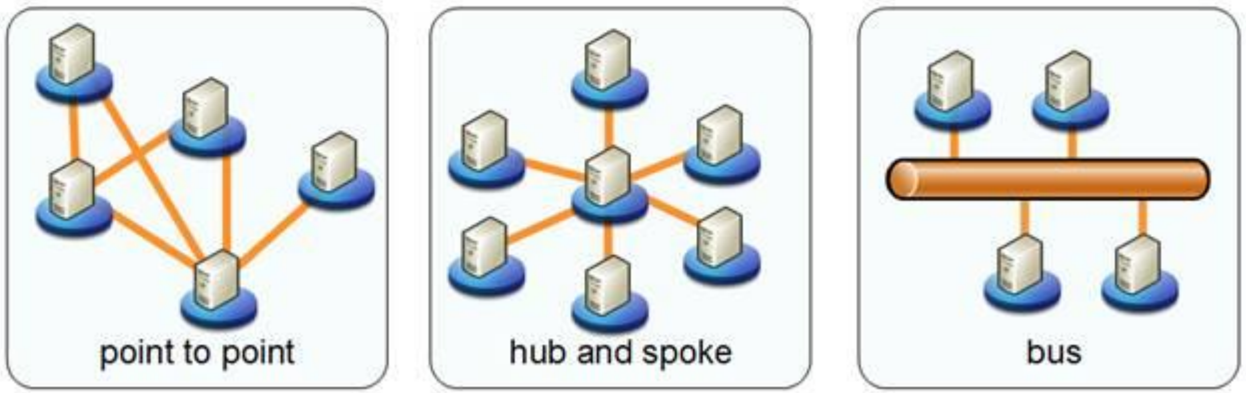
- Simpler connection
- Dedicated line for the connection between the I/O module and peripheral.
- Offers a higher degree of parallelism and greater reliability.
- RTS/CTS protocol (Request/Clear To Send).



3) Connection Sharing Type

b) **Multipoint**

- Shares a set of lines among several peripherals.
- Greater scalability than point-to-point.
- Does not allow parallelism – Scheduling, token, etc.
- Used for storage.
- Examples: IDE, SCSI, USB, etc.



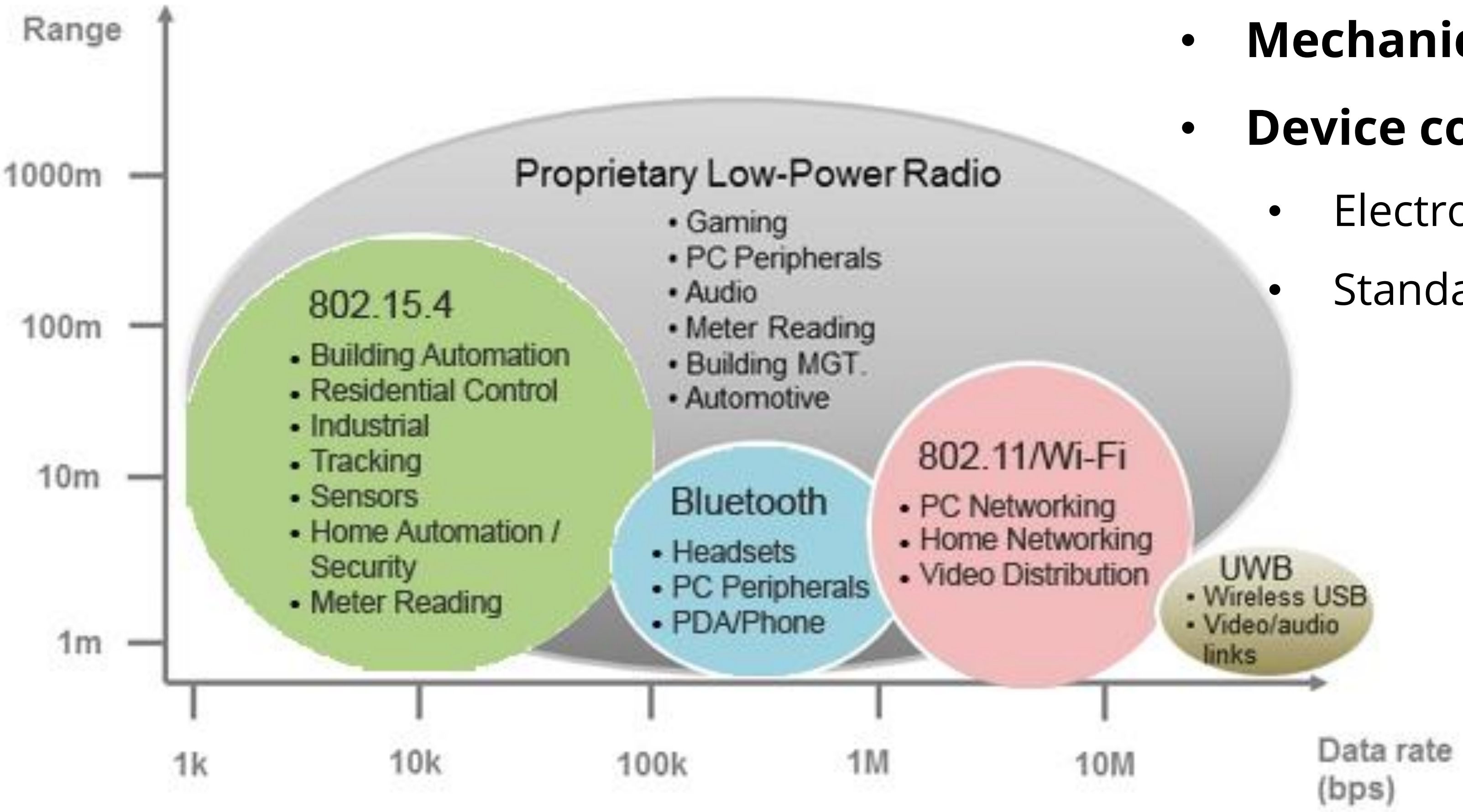
I/O System

- Hardware principles

I/O System
Patterns

Hardware Principles - The I/O units

- **Mechanical component** – the device.
- **Device controller**:
  - Electronic component – Programmable part.
  - Standardization bodies: IEEE, ISO, ANSI, etc.



I/O System

Input type	Prime examples	Other examples	Data rate (b/s)	Main uses
Symbol	Keyboard, keypad	Music note, OCR	10s	Ubiquitous
Position	Mouse, touchpad	Stick, wheel, glove	100s	Ubiquitous
Identity	Barcode reader	Badge, fingerprint	100s	Sales, security
Sensory	Touch, motion, light	Scent, brain signal	100s	Control, security
Audio	Microphone	Phone, radio, tape	1000s	Ubiquitous
Image	Scanner, camera	Graphic tablet	1000s-10^6s	Photos, publishing
Video	Camcorder, DVD	VCR, TV cable	1000s-10^9s	Entertainment
Output type	**Prime examples**	**Other examples**	**Data rate (b/s)**	**Main uses**
Symbol	LCD line segments	LED, status light	10s	Ubiquitous
Position	Stepper motor	Robotic motion	100s	Ubiquitous
Warning	Buzzer, bell, siren	Flashing light	A few	Safety, security
Sensory	Braille text	Scent, brain stimulus	100s	Personal assistance
Audio	Speaker, audiotape	Voice synthesizer	1000s	Ubiquitous
Image	Monitor, printer	Plotter, microfilm	1000s	Ubiquitous
Video	Monitor, TV screen	Film/video recorder	1000s-10^9s	Entertainment
Two-way I/O	**Prime examples**	**Other examples**	**Data rate (b/s)**	**Main uses**
Mass storage	Hard/floppy disk	CD, tape, archive	10^6s	Ubiquitous
Network	Modem, fax, LAN	Cable, DSL, ATM	1000s-10^9s	Ubiquitous

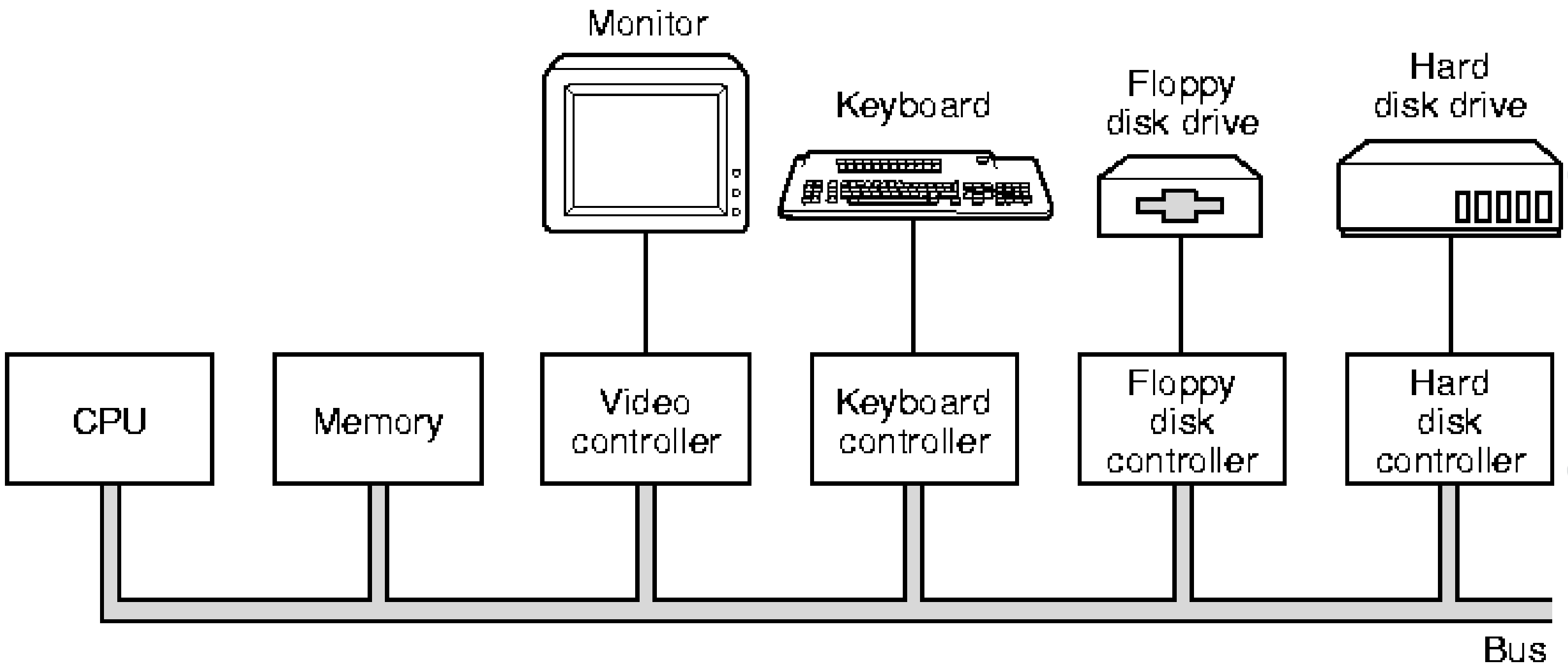
I/O System

Hardware principles

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

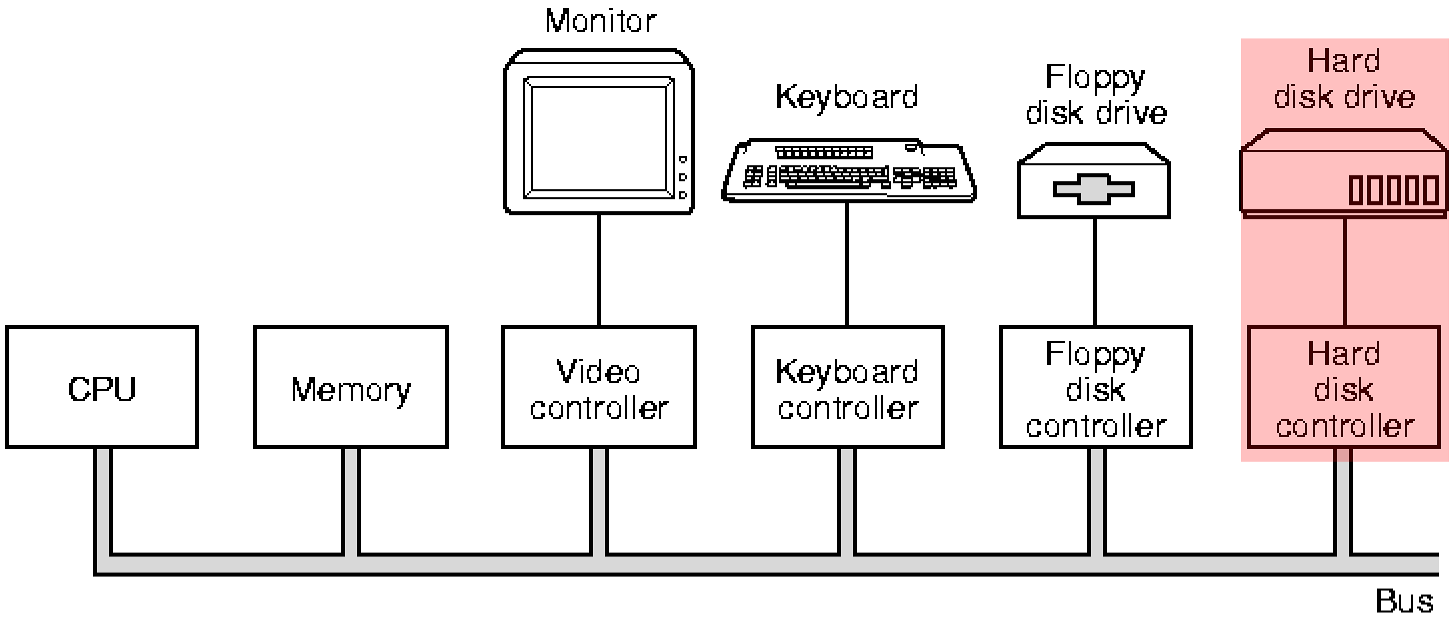
Hardware Principles

- The OS deals with the controller – Does not handle the devices.
- CPU and Controllers Communication:
    - High-level interface – uses common **buses**.
    - Low-level interface – between **controller** and **device**



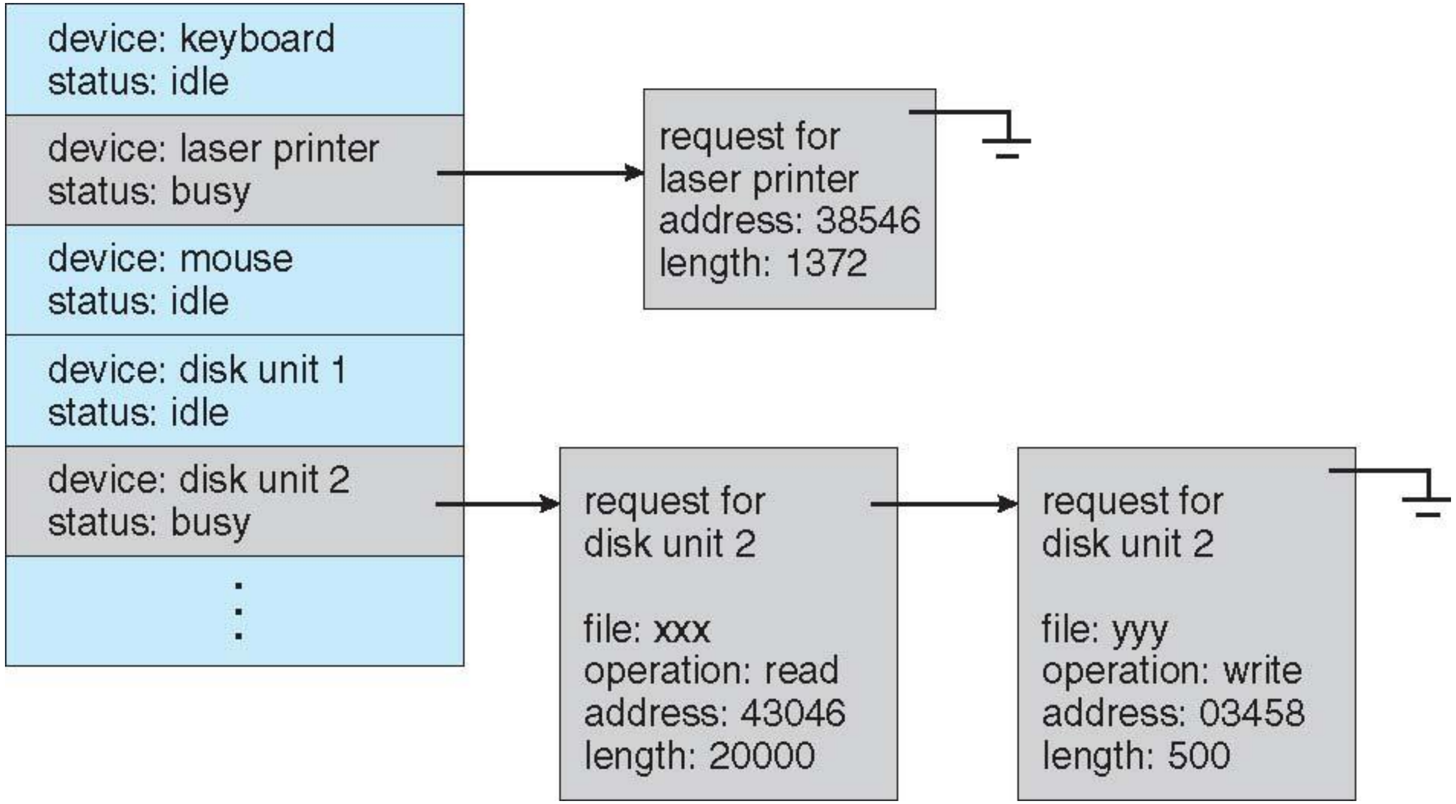
Hardware Principles

- The OS deals with the controller – Does not handle the devices.
- CPU and Controllers Communication:
    - High-level interface – uses common **buses**.
    - Low-level interface – between **controller** and **device**



Hardware Principles - Controllers

- Each controller has **registers** controlled by the CPU to read/write data on the device.
- The OS can control the device by writing **commands** and changing **parameters**.
- Registers are used to know the **device's state**.



1) Memory-mapped

- Located within the memory address space.
- Uses a set of reserved addresses.
- Registers are treated as memory positions.
- All registers are **mapped** in the memory space.
- Generally located at the top of the addressing space.

2) I/O-mapped

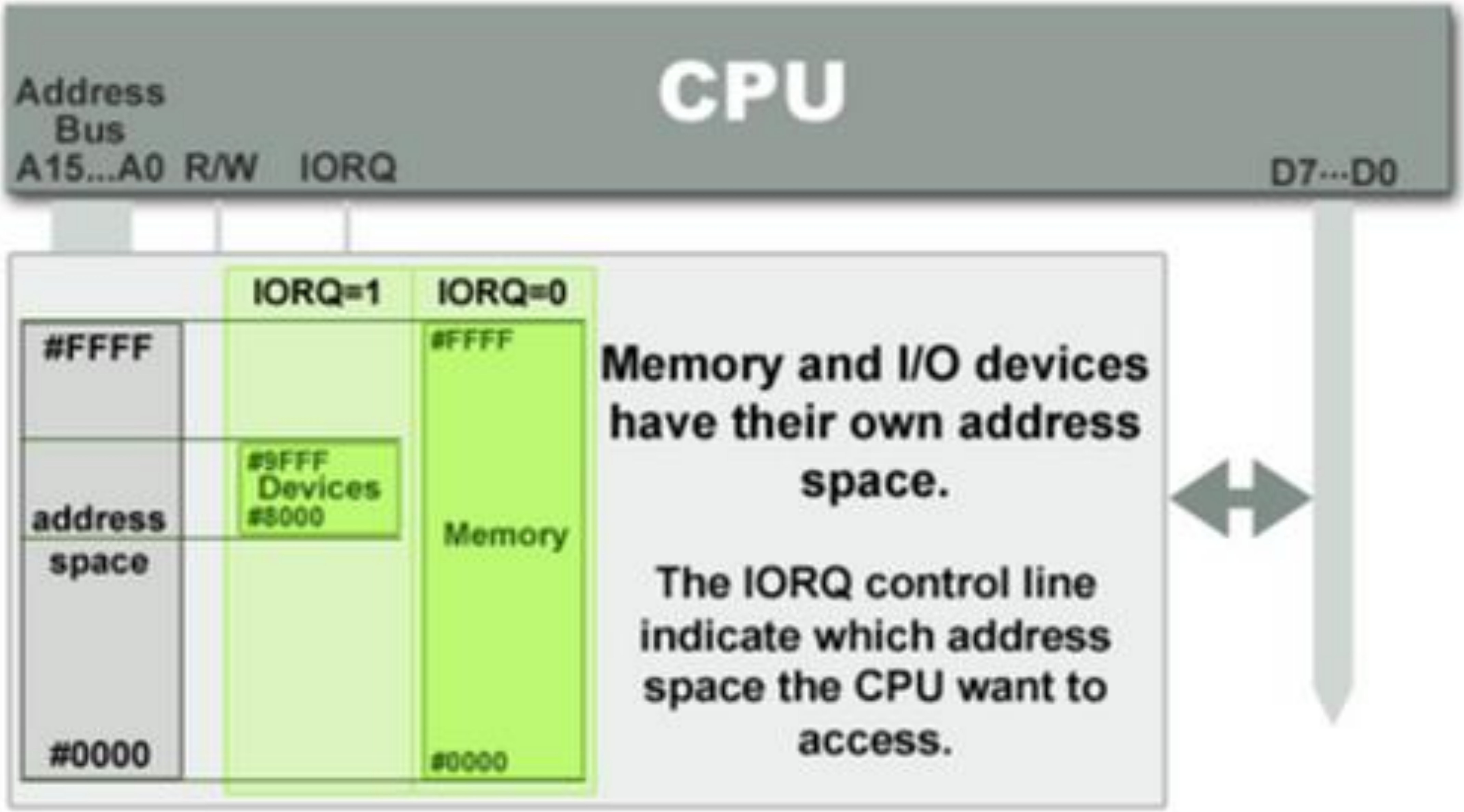
- The controller receives a number of I/O ports accessed via special instructions used only by the OS.
- For example:

```
IN REG, PORT
OUT PORT, REG
```

- Different address spaces for memory and I/O.

3) Hybrid Mapping

- Data buffers in memory;
- I/O ports for control.
- Example – Pentium:
    - 640K to 1M-1 for data buffer
    - I/O ports from 0 to 64K-1 for control



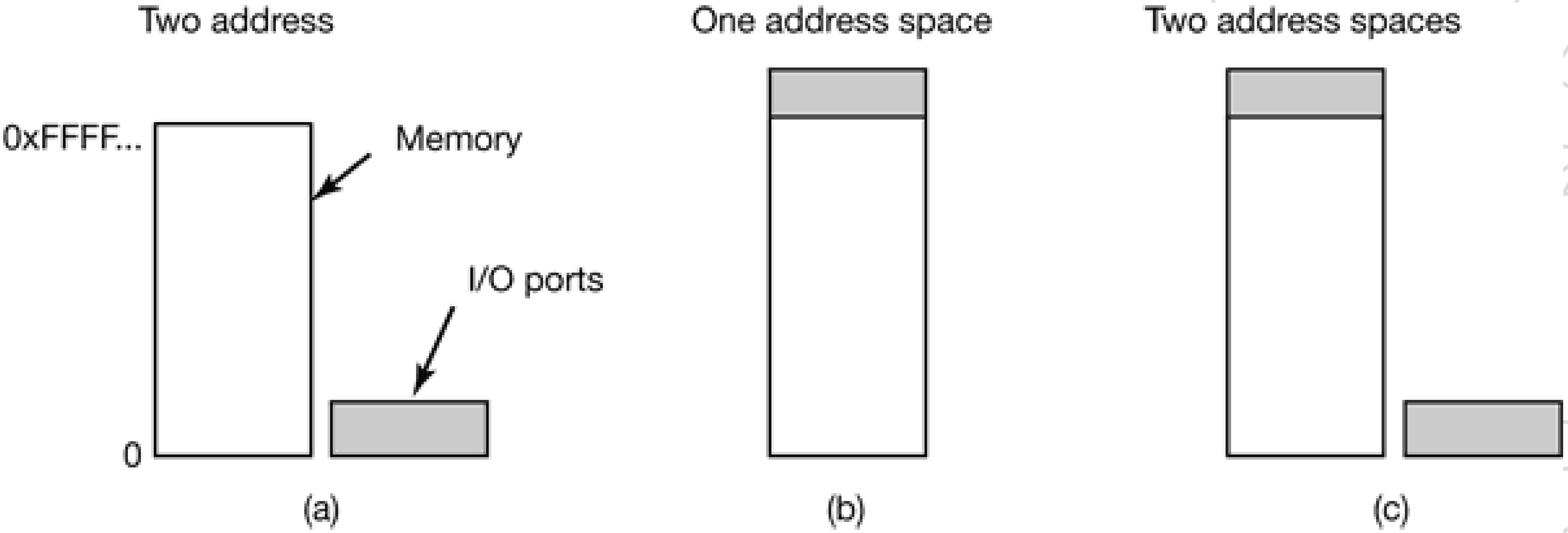
CPU
Address Bus
A15...A0
R/W
IORQ
D7···D0
IORQ=1
IORQ=0
#FFFF
address space
#0000
#9FFF
Devices
#8000
#FFFF
Memory
#0000
Memory and I/O devices have their own address space.
The IORQ control line indicate which address space the CPU want to access.

CPU x Controller Communication Modes

a) I/O-mapped

b) Memory-mapped

c) Hybrid Mapping



I/O System

- Software principles

Software Principles

- **Device independence** (CD, HDD, Flash Drive, etc.) – it's up to the OS to handle the specifics.
- **Patterns** for nomenclature – device-independent names.
- **Error handling** - Should be performed as close as possible to the HW, without the user's knowledge.
    - If a read error occurs, it should be repeated so that the error is not disclosed to the upper layers.

I/O Operation Modes

1. Programmed I/O
2. Interrupt-driven I/O
3. Direct Memory Access (DMA) I/O

What distinguishes the three forms?

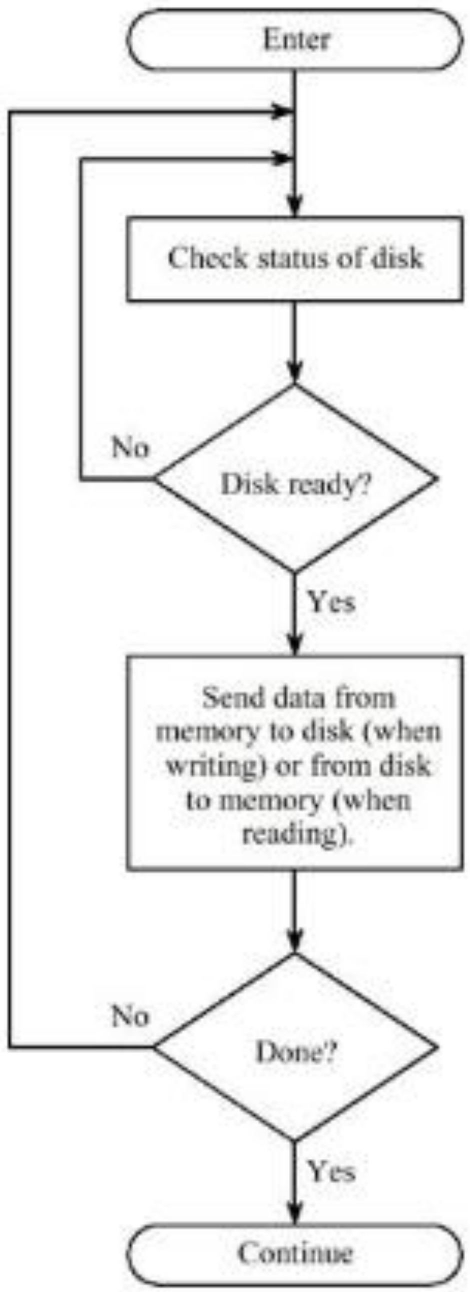
- CPU involvement
- Use of interruptions

1) Programmed I/O

- The simplest form of I/O – everything is done by the CPU.
- Data are exchanged between the CPU and the I/O module.

The CPU executes a program that:

1. Checks the state of the I/O module;
2. Sends the operation command;
3. Waits for the result; (busy waiting)
4. Performs the transfer to the CPU's register.



1) Programmed I/O

Disadvantages

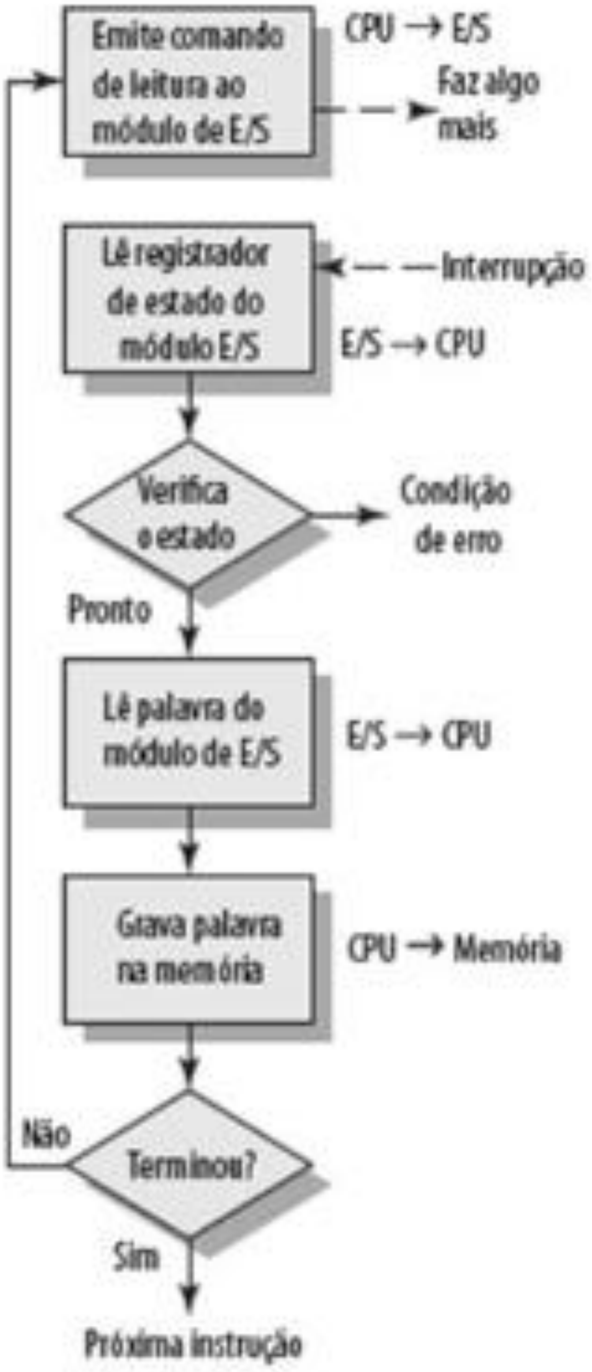
- CPU is occupied all the time and performs all the I/O (lock, turn, TSL).
- Engages in busy waiting for the task to be completed, also called polling.

2) Interrupt-driven I/O

- Overcomes the problem of busy waiting.
- Interruptions are identified by numbers - The smaller, the higher priority.

The CPU executes a program that:

1. Sends an I/O command;
2. CPU performs another operation;
3. Controller sends a signal when I/O is finished;
4. CPU reads the data from the controller.



2) Interrupt-driven I/O

Example of an interruption:

1) CPU requests a read on the disk;
2) Controller reads the data while the CPU performs other tasks;
3) Controller sends an interruption to the CPU;
4) CPU requests the data;
5) Controller sends the data.

I/O System

Interrupt-driven mode

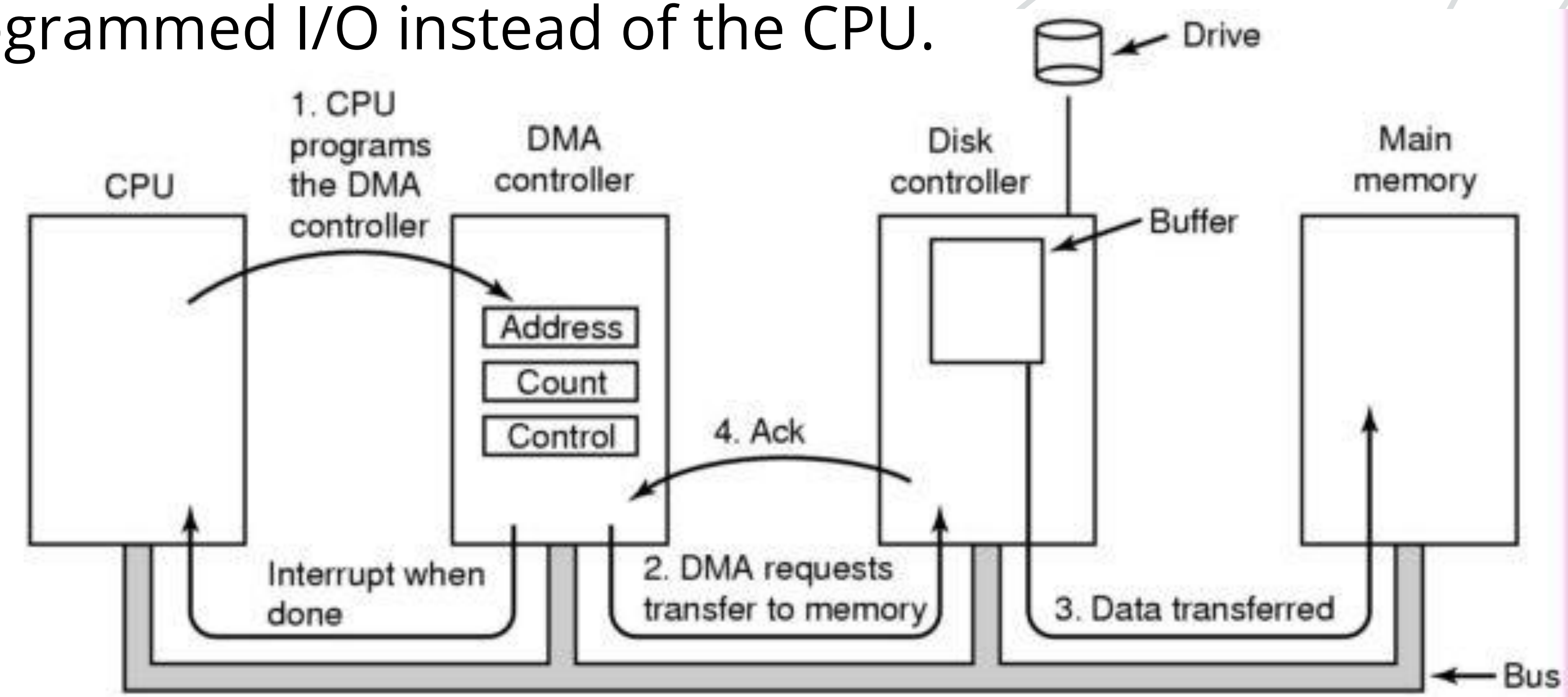
Below is a table showing the Interrupt Numbers and Names.

INT #	Normal Use	INT #	Normal Use	INT #	Normal Use
0	Divide by Zero (Internal)	12	BIOS Get Memory Size	24	Critical Error Handler *
1	Single Step Debug	13	BIOS Diskette Service	25	DOS Absolute Disk Read *
2	NMI *	14	BIOS Comm. Services *	26	DOS Absolute Disk Write *
3	Breakpoint *	15	BIOS Misc. System Services *	27	Terminate and Stay Resident (TSR)
4	Arithmetic Overflow	16	BIOS Keyboard Services	28	DOS safe *
5	Print Screen *	17	BIOS Printer Services	29	DOS TTY
6	Invalid Opcode	18	Execute	2A	MS Net
7	CPU Reserved	19	System Warm Reboot *	2F	"Multiplex" *
8	System Timer *	1A	BIOS Clock Services	33	Microsoft Mouse Services
9	Hardware Keyboard	1B	Ctrl-Break Handler *	67	EMS Services
A	Cascade to IRQ 9	1C	User Timer Tick Interrupt	70	Real Time Clock
B	COM 2*	1D	Video Init. Parameters	71	Redirect to IRQ 2
C	COM 1*	1E	Disk Init. Parameters *	72	USER HARDWARE
D	LPT 2	1F	Grap Display Char Table*	73	USER HARDWARE
E	Floppy Diskette Controller	20	DOS Terminate Program	74	IBM Mouse (Hardware) *
F	LPTI	21	DOS Services	75	Math Coprocessor
10	BIOS Video Services	22	DOS Termination Address	76	Hard Disk Controller
11	BIOS Get Equipment Status	23	CtrlC Handler *	77	USER HARDWARE

IRQ	Usage
0	system timer (cannot be changed)
1	keyboard controller (cannot be changed)
2	cascaded signals from IRQs 8–15
3	second RS-232 serial port (COM2: in Windows)
4	first RS-232 serial port (COM1: in Windows)
5	parallel port 2 and 3 or sound card
6	floppy disk controller
7	first parallel port
8	real-time clock
9	open interrupt
10	open interrupt
11	open interrupt
12	PS/2 mouse
13	math coprocessor
14	primary ATA channel
15	secondary ATA channel

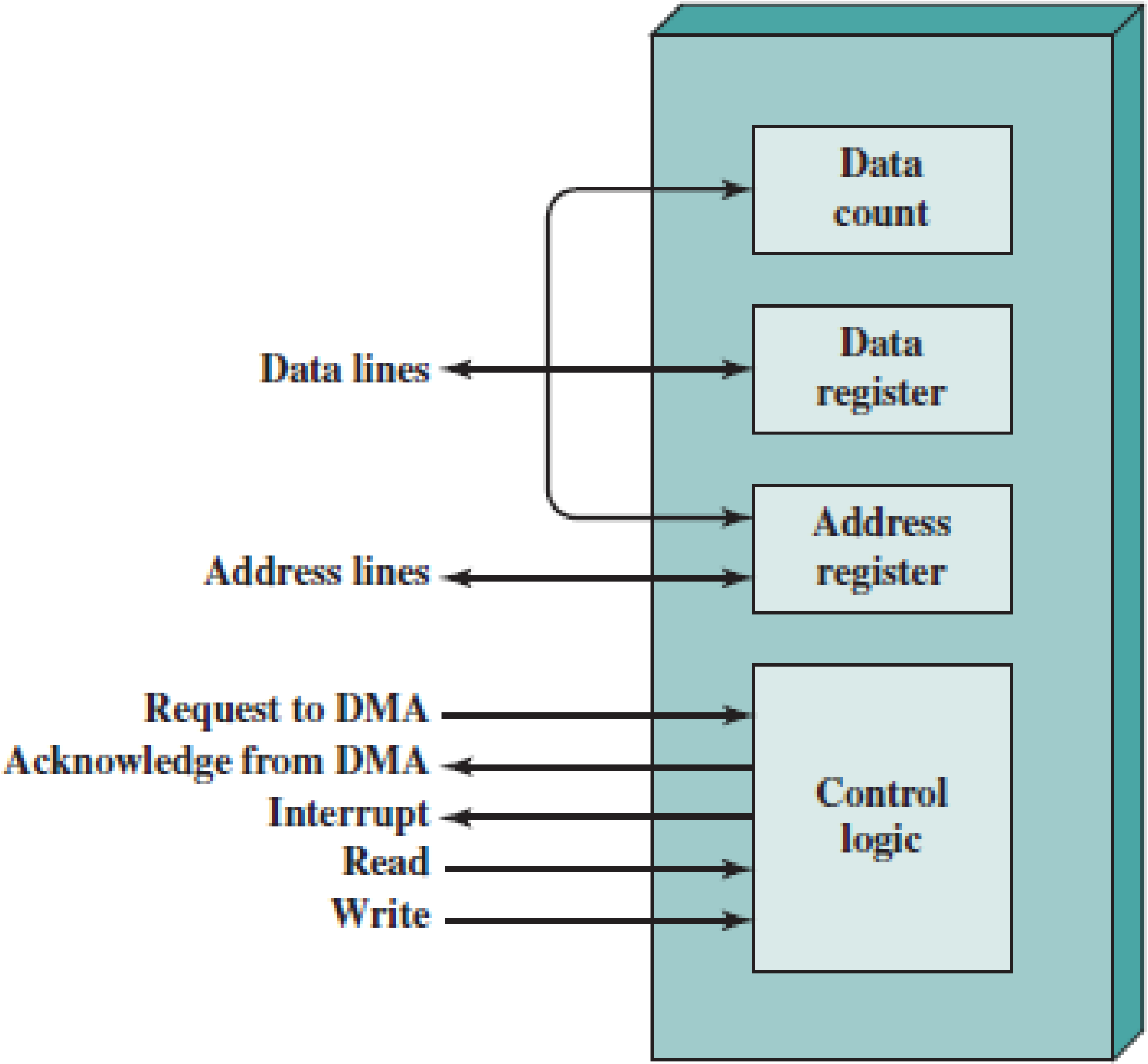
3) DMA I/O

- Requires software and hardware.
- This solution removes the CPU from management but needs a DMA (Direct Memory Access) controller.
- The DMA performs Programmed I/O instead of the CPU.



3) DMA I/O

- Requires software and hardware.
- This solution removes the CPU from management but needs a DMA (Direct Memory Access) controller.
- The DMA performs Programmed I/O instead of the CPU.
- Disadvantages of previous techniques:
  - Limit the CPU's transfer capacity;
  - CPU is busy managing;
  - Performance drops for large amounts of data.



3) DMA I/O

Necessary information:

a) Memory address;

b) Amount of bytes;

c) I/O port to be used;

d) Direction of transfer (from or to the device)

e) Transfer unit (one byte or word at a time)

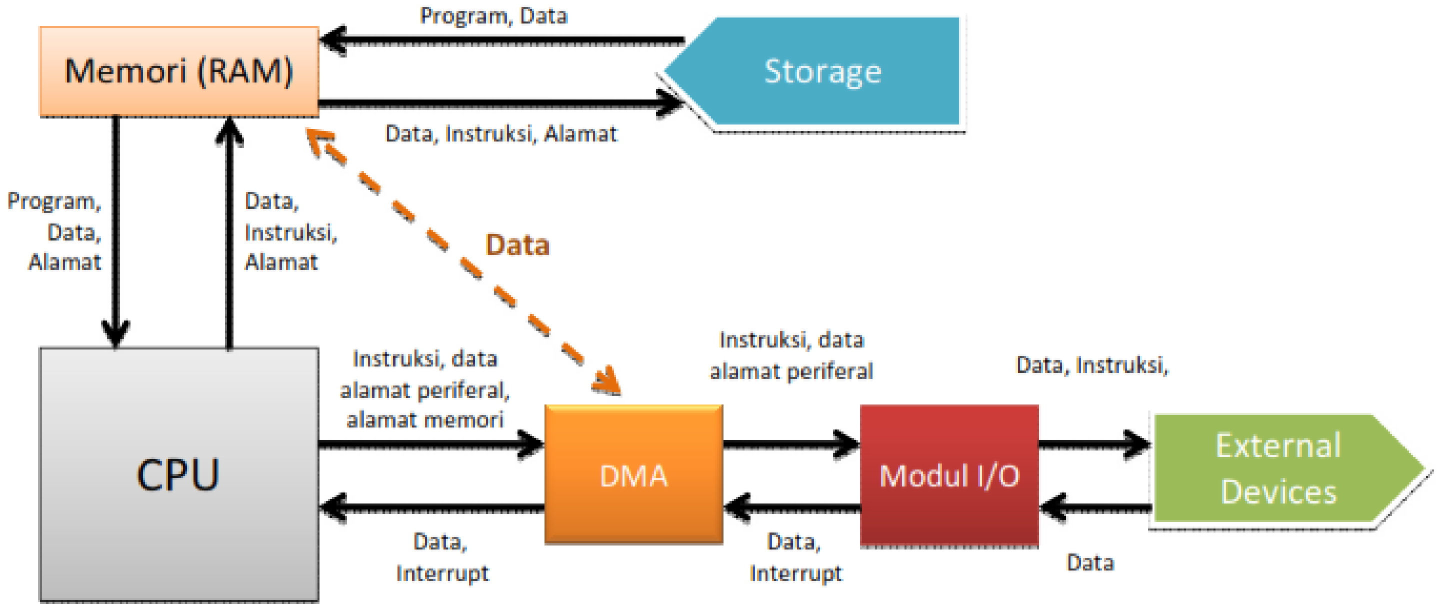
3) DMA I/O

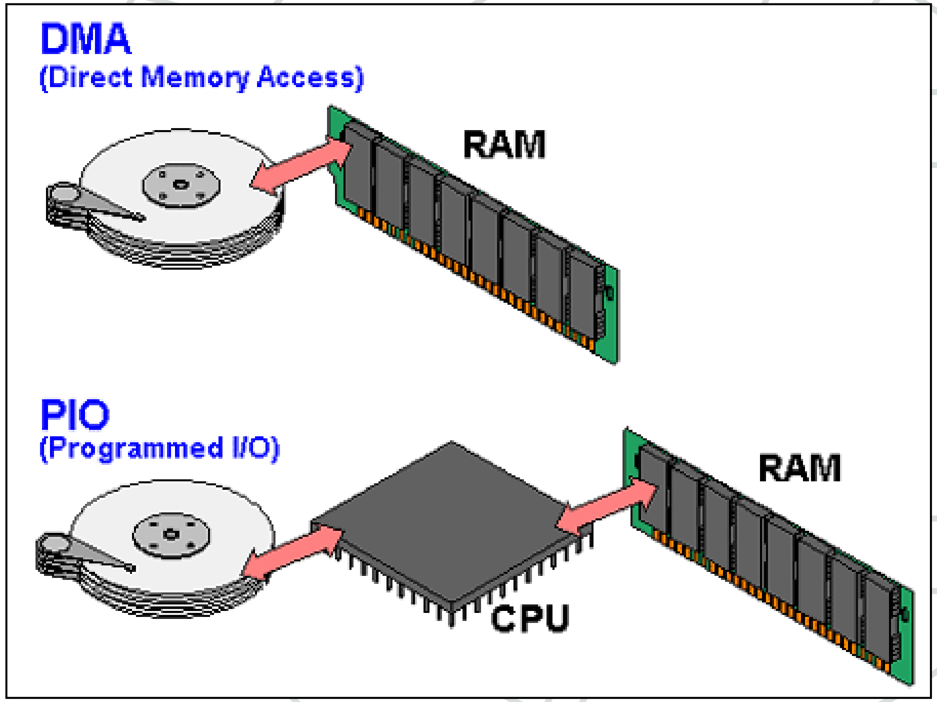
Disadvantages:

- The CPU may be faster than the DMA controller.
- More expensive architecture with DMA.

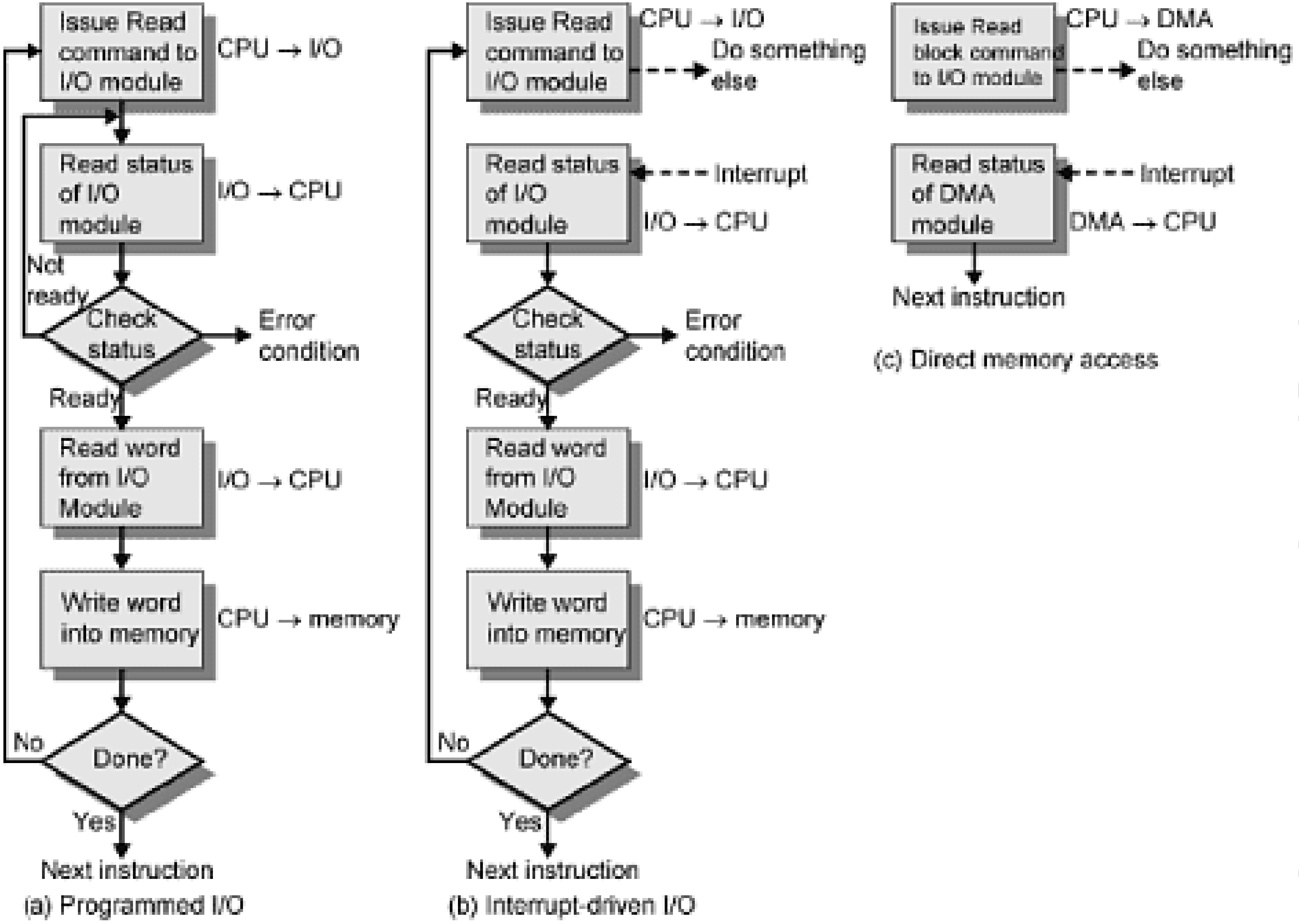
Advantages:

- DMA performs Programmed I/O.
- The DMA controller does all the work and frees up the CPU.





I/O System


Issue Read command to I/O module
CPU → I/O
Read status of I/O module
I/O → CPU
Not ready
Check status
Error condition
Ready
Read word from I/O Module
I/O → CPU
Write word into memory
CPU → memory
No
Done?
Yes
Next instruction
(a) Programmed I/O
Issue Read command to I/O module
CPU → I/O
Do something else
Read status of I/O module
Interrupt
I/O → CPU
Check status
Error condition
Ready
Read word from I/O Module
I/O → CPU
Write word into memory
CPU → memory
No
Done?
Yes
Next instruction
(b) Interrupt-driven I/O
Issue Read block command to I/O module
CPU → DMA
Do something else
Read status of DMA module
Interrupt
DMA → CPU
Next instruction
(c) Direct memory access

Layer Principle

Layer Principle

Facilitates device **independence**, providing **modularity** and **cohesion**.

Lower layers:

- Hardware details;
- Drivers and interrupt handlers.

Higher layers:

- User interface;
- User applications;
- System calls – I/O-independent part.

Device Independence

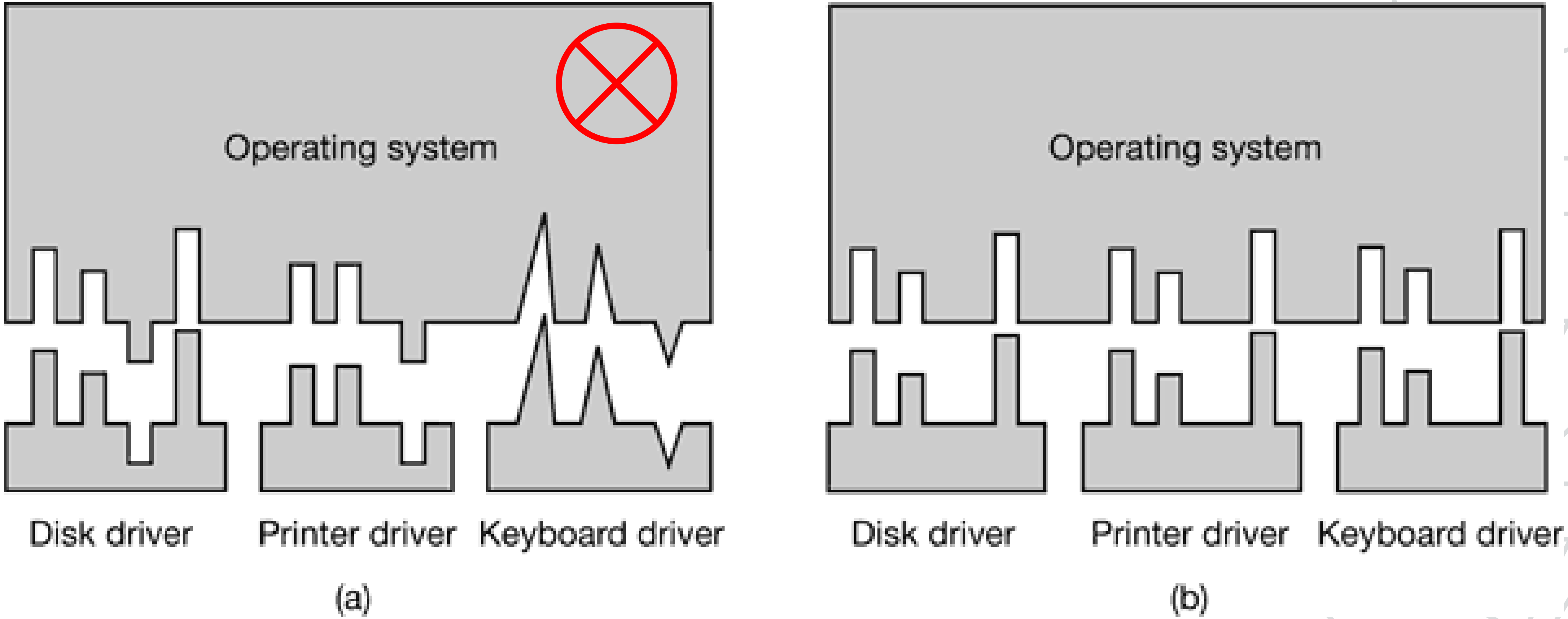
- Provide a uniform interface to user software.
- Avoid the OS having to be modified with each new device created.
- I/O software – there are device-specific parts and device-independent parts.
- The independent part:
  - Performs I/O common to all devices;
  - Performs I/O scheduling;
  - Provides buffering – adjusts speed and the amount of data transferred.
  - Provides data caching – stores a set of frequently accessed data..

Device Independence

- Other functions of the independent part:
  - Report errors and protect against improper access:
    - Programming errors – Ex.: reading from an output device (video).
    - I/O errors – Ex.: printing on a printer without paper.
    - Memory errors – Ex.: writing to invalid addresses (segmentation fault).
  - Define device-independent block sizes.

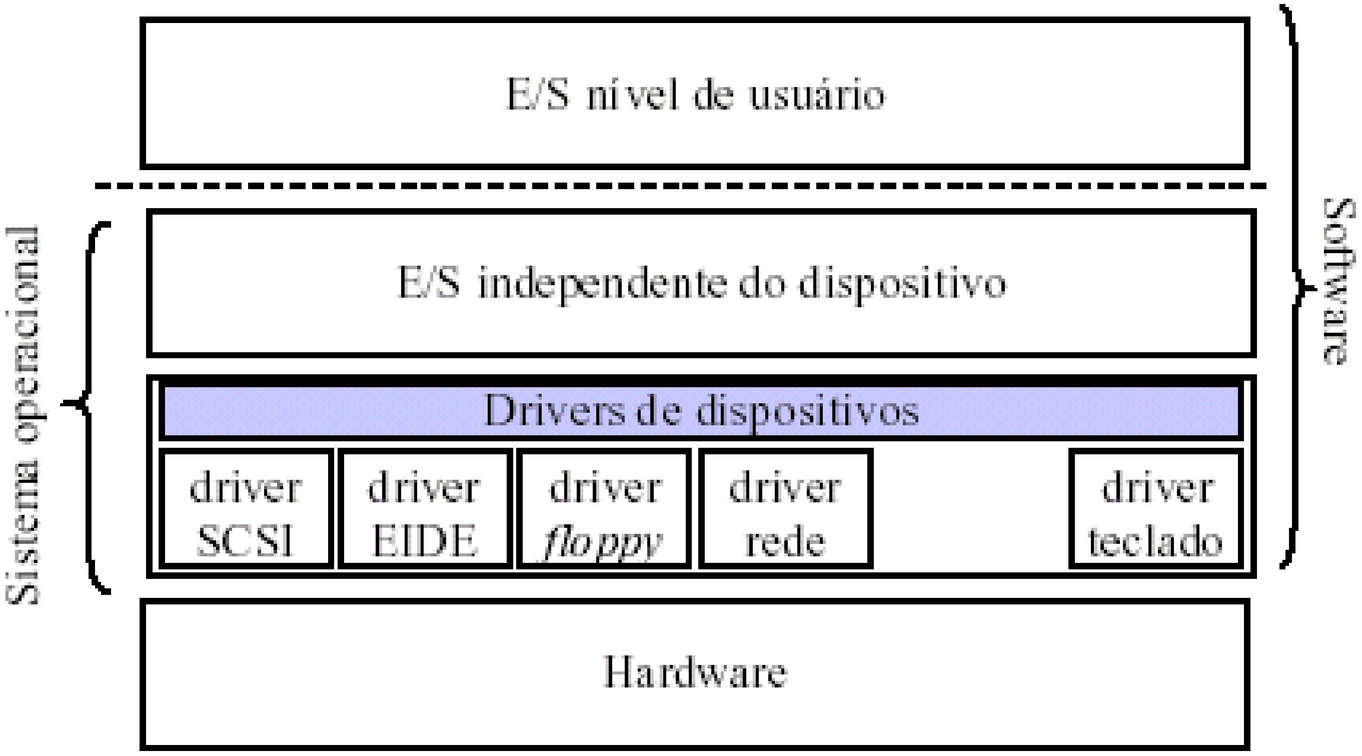
Software Principles

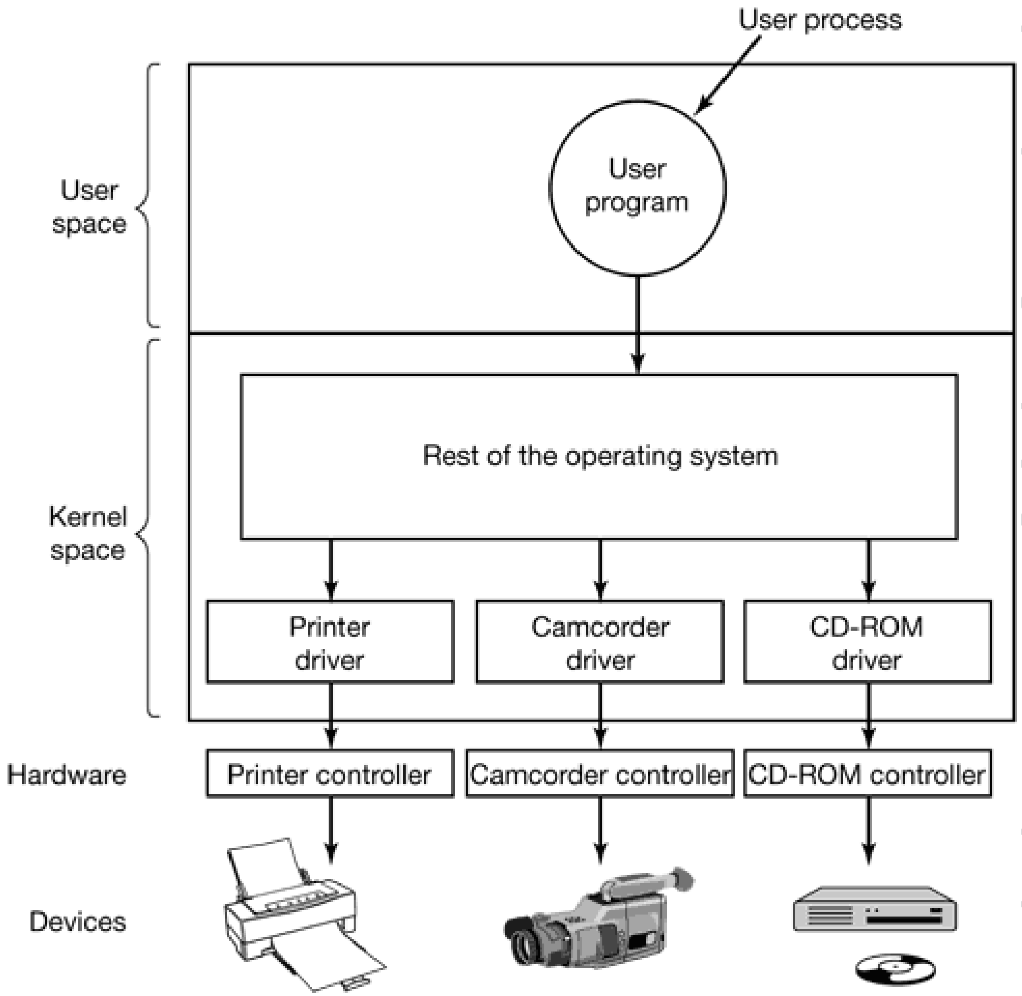
- Provide a uniform interface (API) – read, write, send, receive, etc.
- Standardize the functions of drivers – each manufacturer provides its function.



***Drivers** – Device Dependent Part*

- *Written by the device manufacturer, according to the defined interface.*

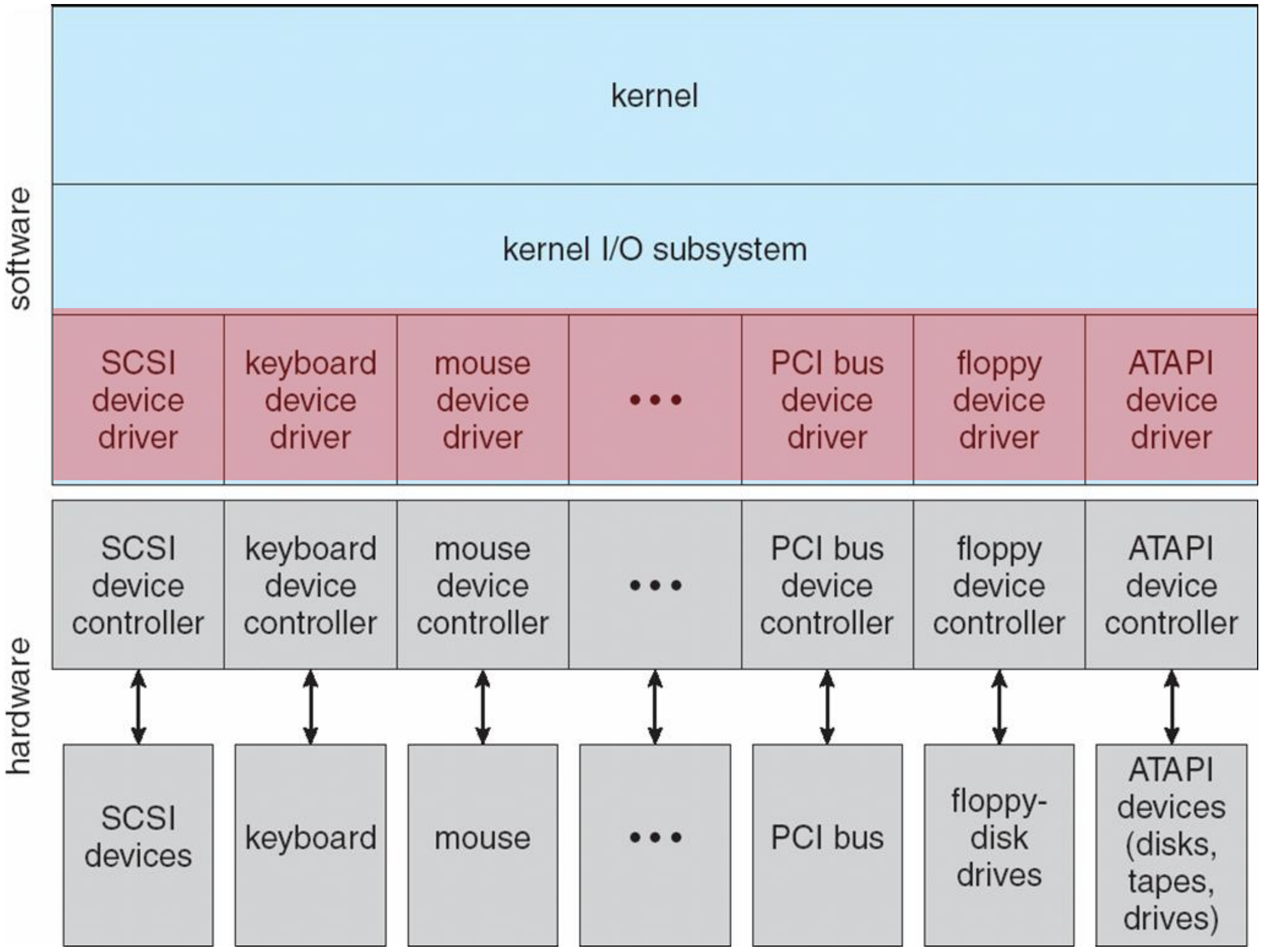




Drivers *– Device Dependent Part*

- *Written by the device manufacturer, according to the defined interface.*
- *Different OSs require different drivers:*
    - *They are part of the kernel and have full access to the device.*
    - *Can cause problems in the OS.*
- *Specific code for I/O and control:*
    - *Winchester disk (HDD) with platters and arm;*
    - *Solid-state drive (SSD).*
- *Usage process (Linux):*
    - *Compile the driver code*
    - *insmod installs the driver object.*
    - *rmmod removes the driver.*

I/O System
Drivers

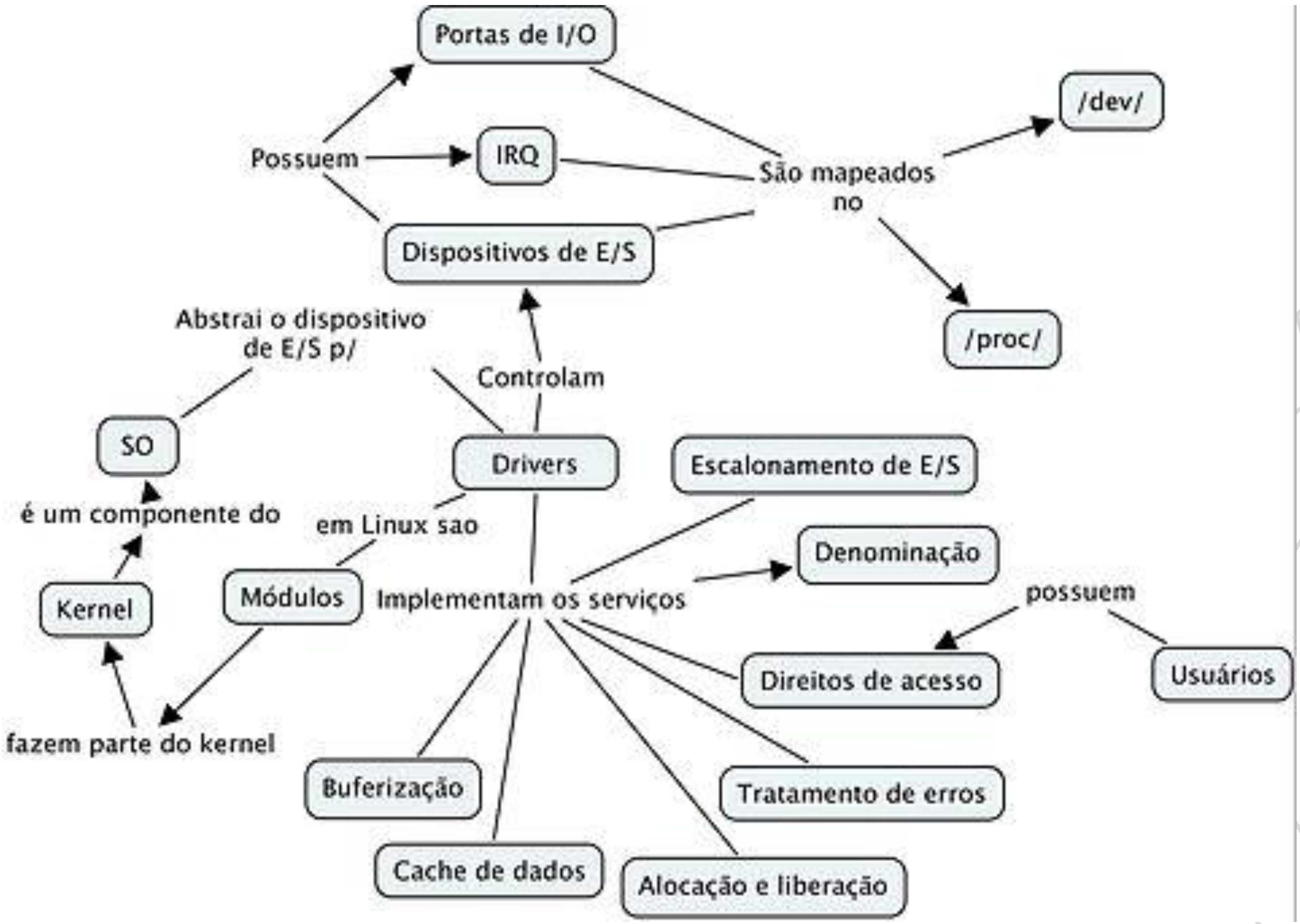


***Drivers** – Device Dependent Part*

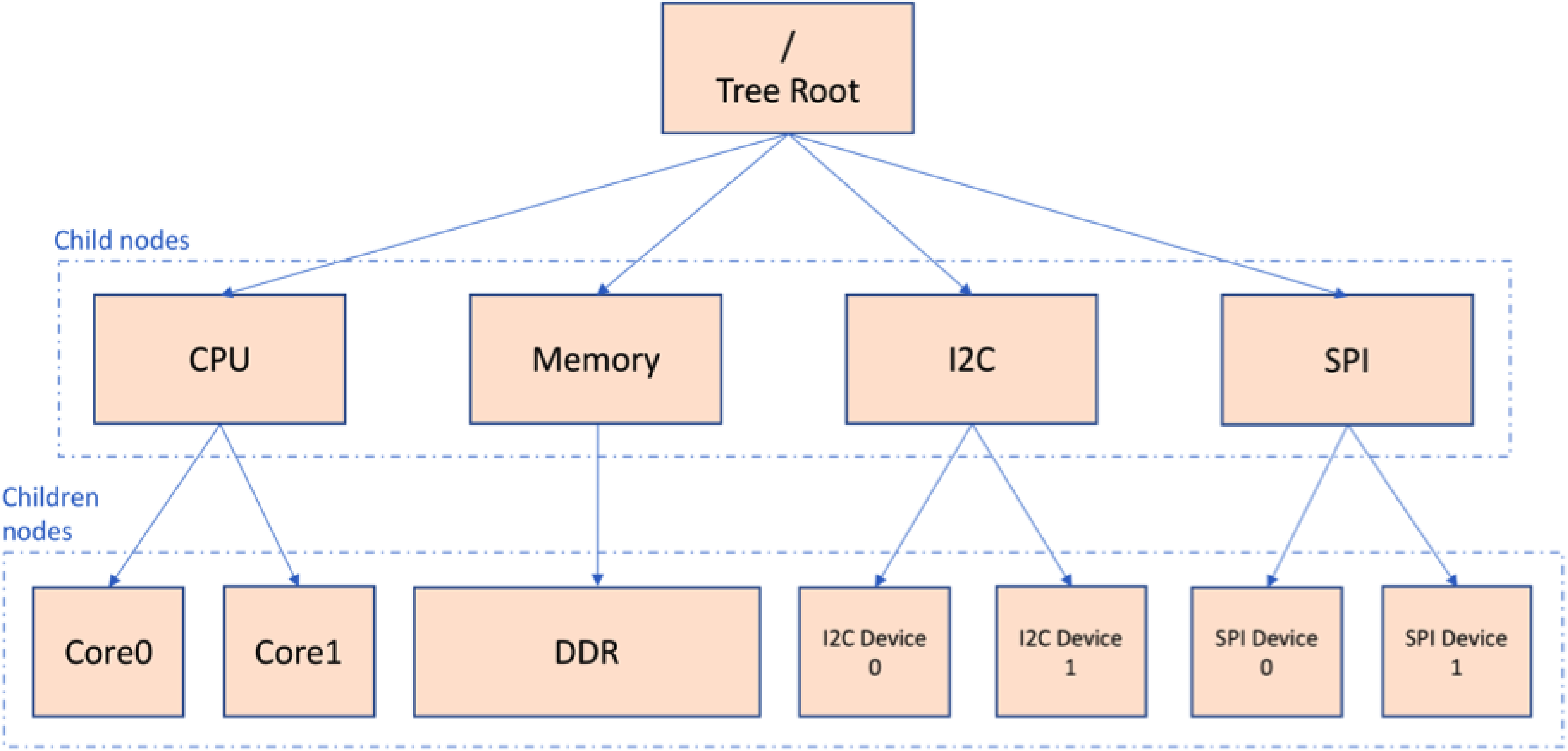
- *Can be dynamically loaded (for example, **DLLs**).*
- *Used for read/write requests made by the software:*
    - *Check the made request - check-up;*
    - *Initialize the device if necessary;*
    - *Manage the device's power needs;*
    - *Create an event log.*

I/O System

Drivers

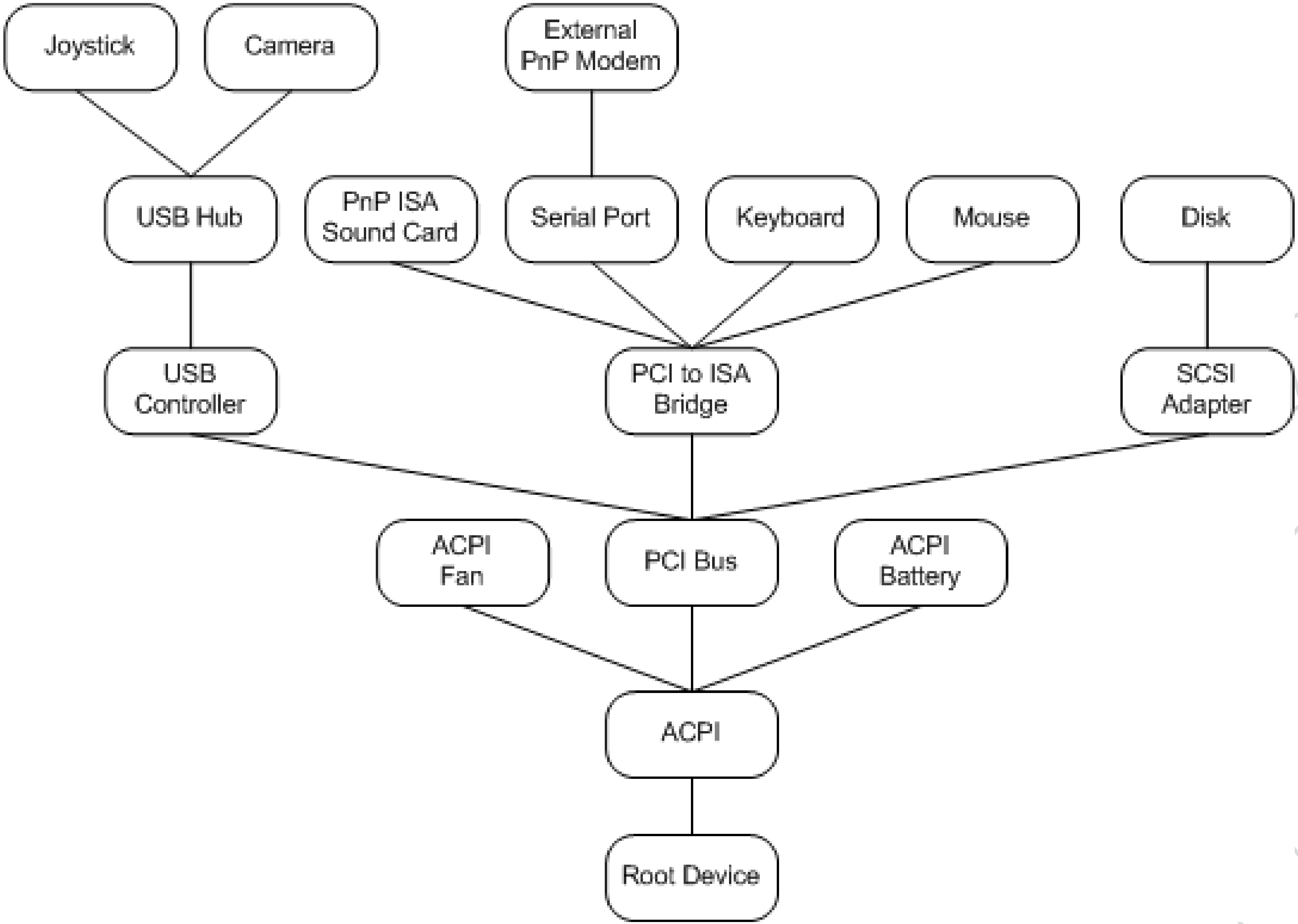


I/O System
Device tree



I/O System

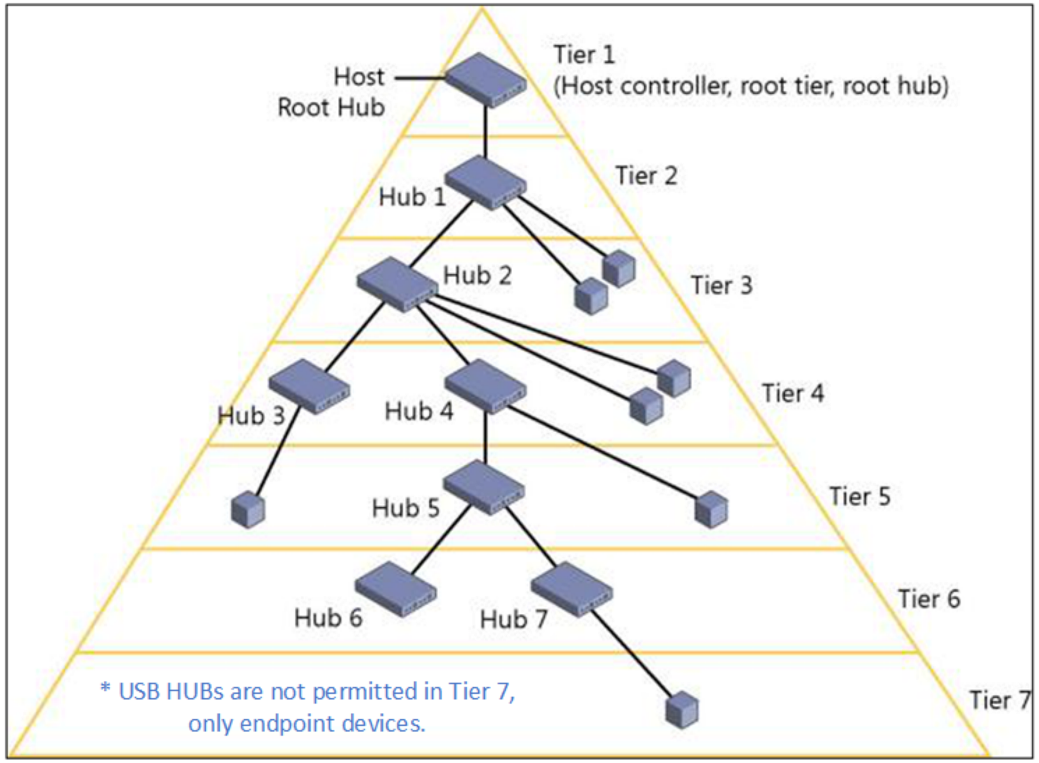
Device tree



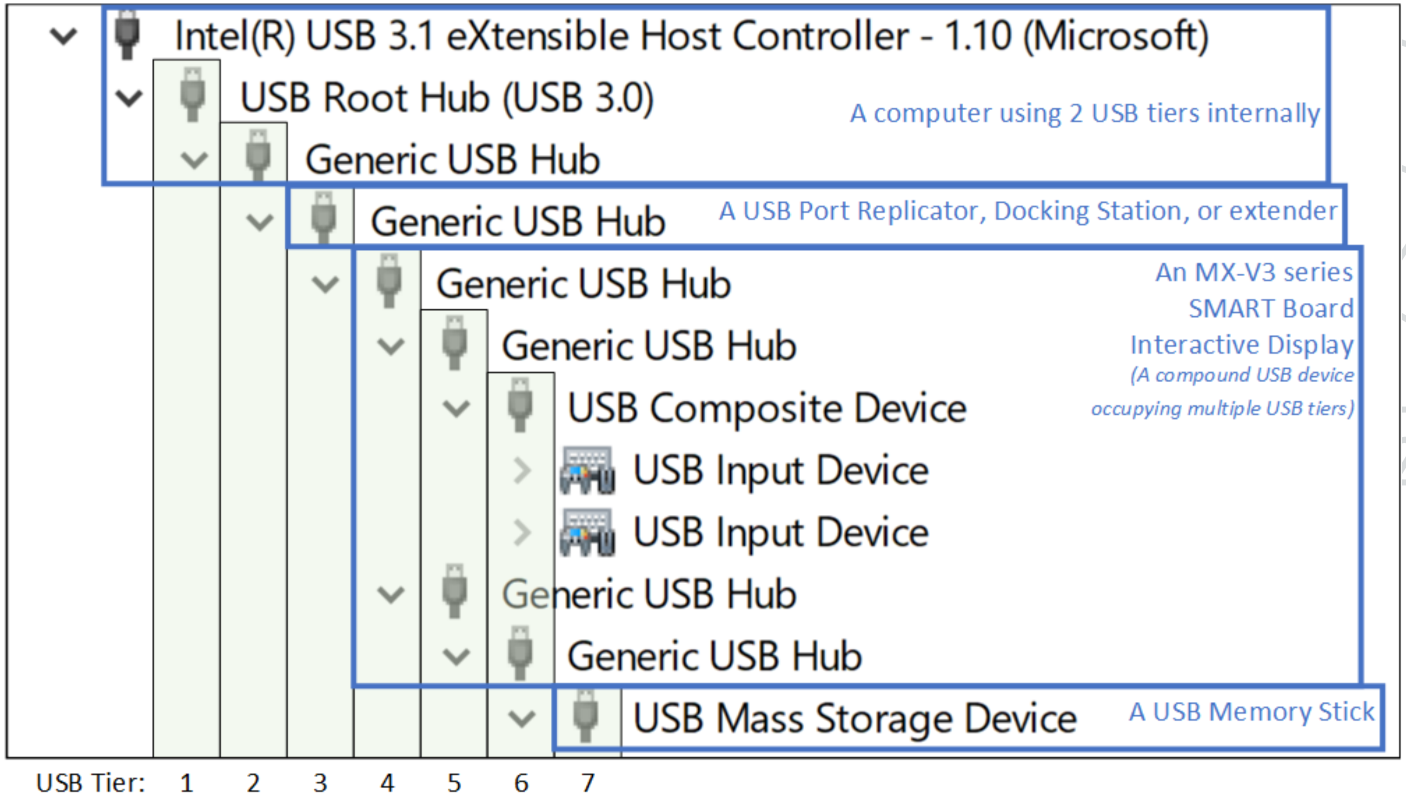
I/O System

USB Device tree

USB Tier Topology
(defined by the USB Implementer's forum)



Windows Device Manager – View Devices by Connection



Drivers

Void pointers

Void pointers

- Points to a memory region **without specifying the type**.
- Can not be used without **casting**.
- Abstraction that allows the programmer to pass the **parameter of different types** to the same function.
- A function that receives **how to know how to handle** each type.

```c
char *name = "Paulo";

double weight = 87.5;

unsigned int children = 3;

void main (void){
    //não confundir com printf
    print(0, name);
    print(1, &weight);
    print(2, &children);
}
```

Void pointers

Void pointers

```c
void print(int option; void *parameter){
  switch(option){
     case 0:
        printf("%s",(char*)parameter);
     break;
     case 1:
        printf("%f",*((double*)parameter));
     break;
     case 2:
        printf("%d",*((unsigned int*)parameter));
     break;
  }
}
```

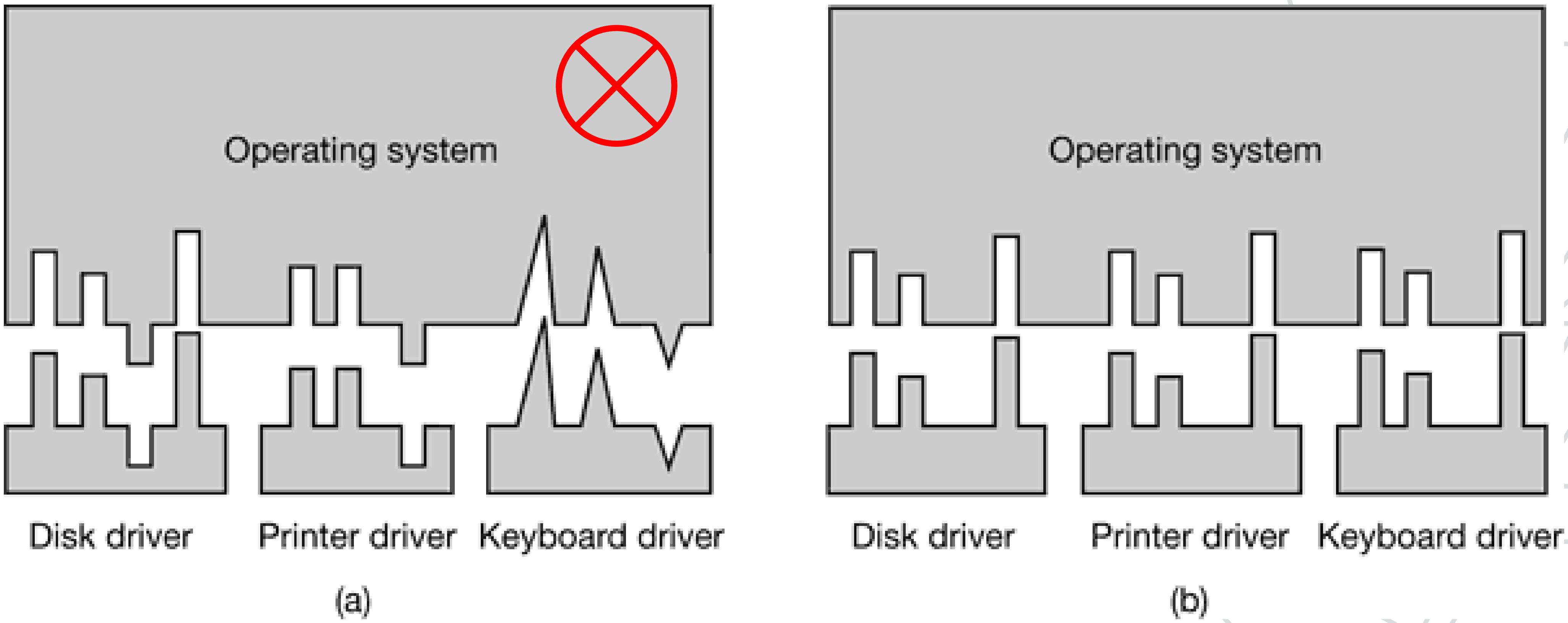
Driver

Driver

- What is a driver?
  - An interface layer that translate **hardware** to **software**
- It have two requirements
  - First: It is **dependent on the microcontroller**, the attached peripherals and the connections between them.
  - Second: Need to comply with the built-in **standard** from the operational system

Driver

- What is a driver?
    - An interface layer that translate **hardware** to **software**



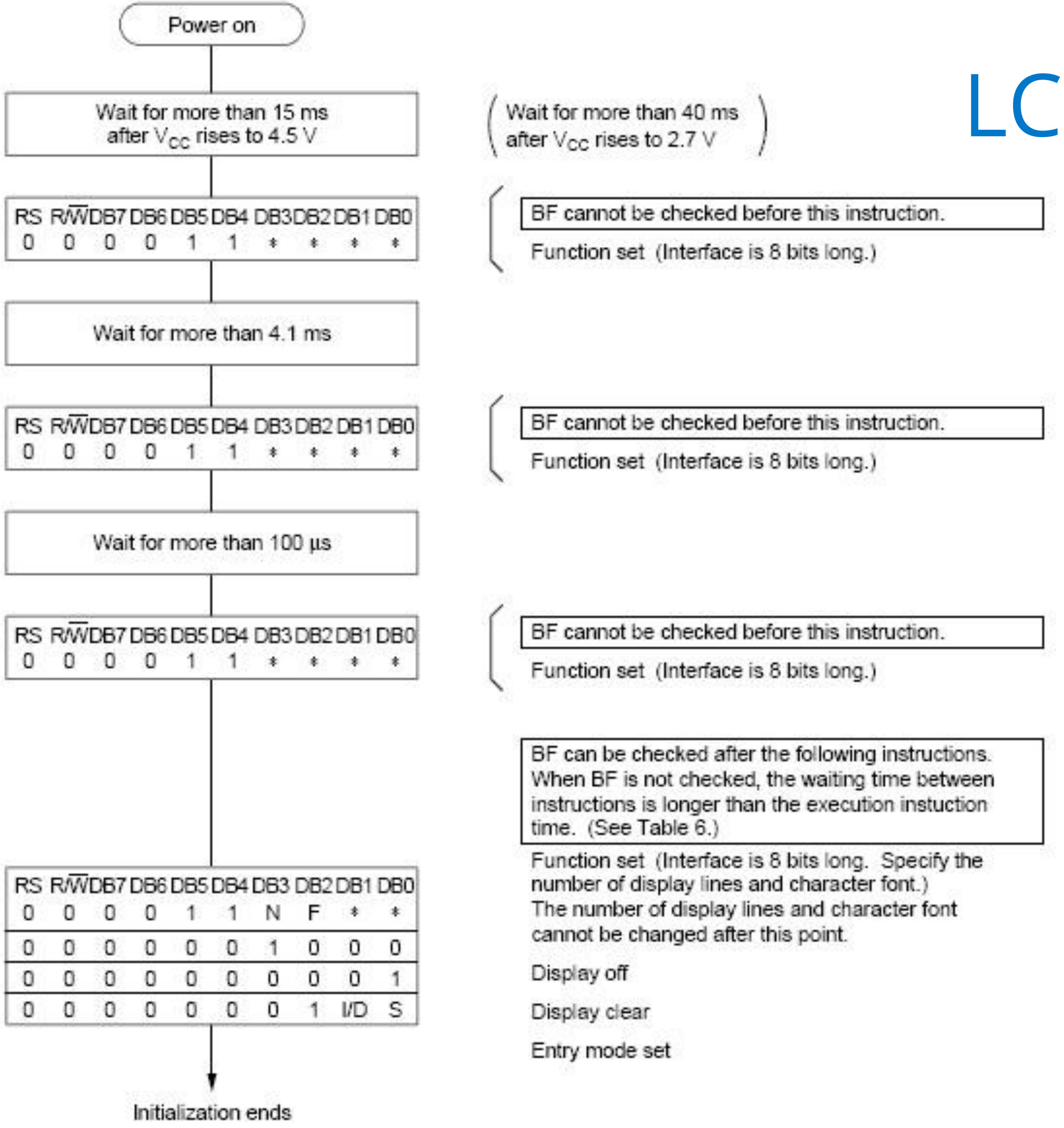
How can I develop my driver?

- First: know your hardware
  - Which features from the microcontroller are needed?
  - How the external peripheral is connected?
  - Are there any software/communication routines required?

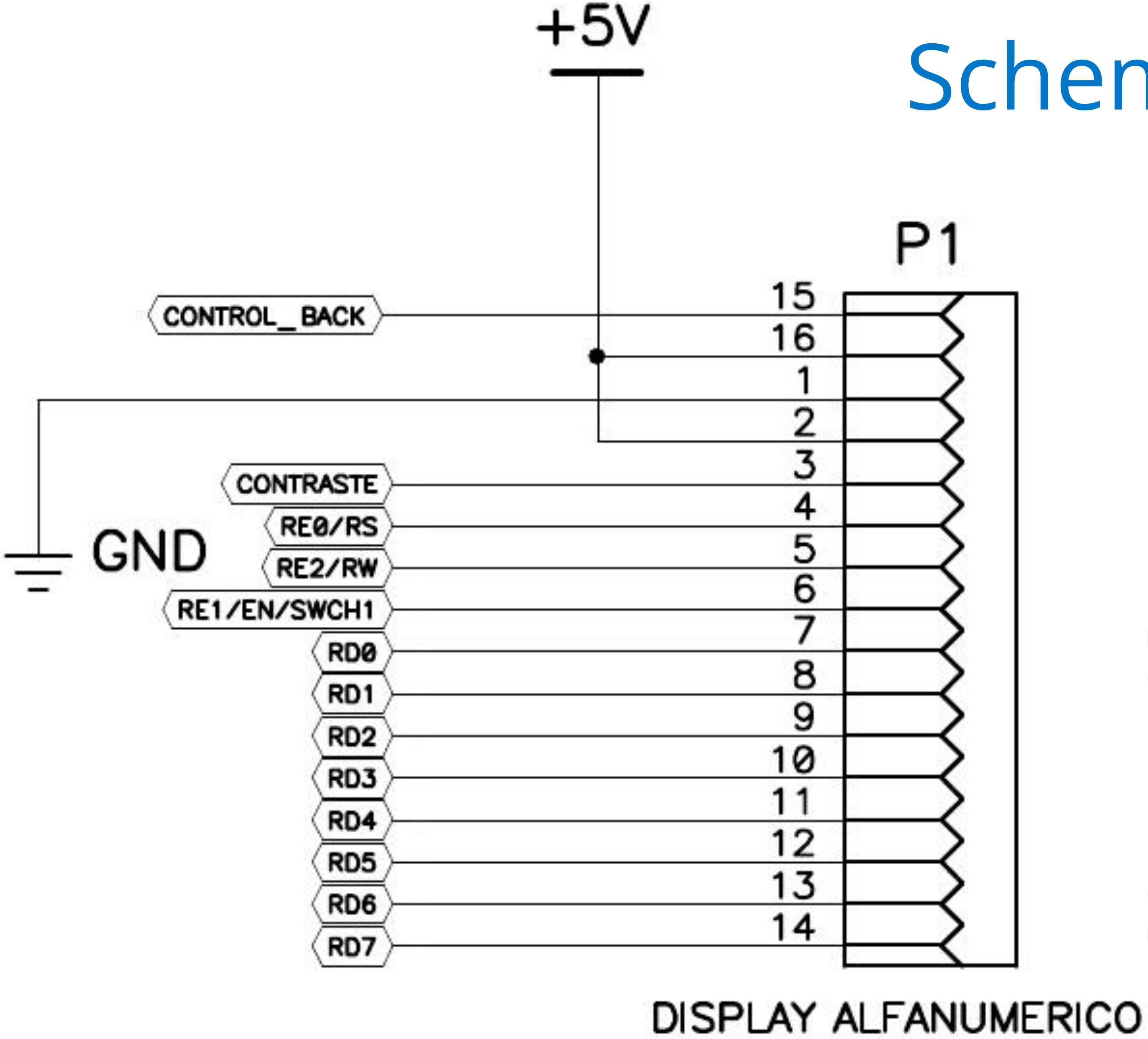
Developing my driver (1)

- Example:
  - LCD Display:
    - 16 columns X 2 lines
    - Compatible with HD44780 (Hitachi)
  - NEO201
    - 8 bit connection (data)
    - Direct access to EN, RS e RW (control)
  - PIC18F4520
    - Using both port D (data) and port E (control)
  - Initialization routines

LCD initialization routine



Schematics LCD - PIC



Basic communication

```c
void lcdData(unsigned char valor){
   BitSet(PORTE,RS);  //data
   BitClr(PORTE,RW);  //write

   PORTD = valor;

   BitSet(PORTE,EN);   //enable pulse
   BitClr(PORTE,EN);

   BitClr(PORTE,RS);  //avoid problems with 7 seg disp
   Delay40us();
}
```

Basic communication

```c
void lcdInit(void){
	// lcd init 10ms
	Delay2ms();
	Delay2ms();
	Delay2ms();
	Delay2ms();
	Delay2ms();

	//pin directions
	BitClr(TRISE,RS);      //RS
	BitClr(TRISE,EN);      //EN
	BitClr(TRISE,RW);      //RW
	TRISD = 0x00;          //data
	ADCON1 = 0b00001110;   //Analog/digital config

	lcdCommand(0x38);      //8bits, 2 lines, 5x8
	lcdCommand(0x06);      //incremental mode
	lcdCommand(0x0F);      //display, cursor , blink on
}
```

How can I develop my driver?

- Second: know your operational system
  - Is there any **standard** on function parameters?
  - Which structures and information do I need to provide to the operational system?
  - How the OS will pass information for me?

Talking with any type of driver

- Parameters problem
  - The kernel must be able to communicate in the same way with all drivers
  - Each function in each driver have different types and quantities of parameters
- Solution: **Pointer to void**

```
void *ptr;
```

Void pointer

Void pointers

- Points to a memory region without specifying the type.
- Can not be used without casting.
- Abstraction that allows the programmer to pass parameters of different types to the same function.
- The function that receives them **must know how to handle** each type.

Void pointers

```c
char *name = "Paulo";

double weight = 61.5;

unsigned int children = 3;

void main (void){
    //it is not printf!!!!
    print(0, name);
    print(1, &weight);
    print(2, &children);
}
```

```c
void print(int option, void *parameter){
  switch(option){
     case 0:
        printf("%s",(char*)parameter);
     break;
     case 1:
        printf("%f",*((double*)parameter));
     break;
     case 2:
        printf("%d",*((unsigned int*)parameter));
     break;
  }
}
```

Void pointer

How to build?

The standard

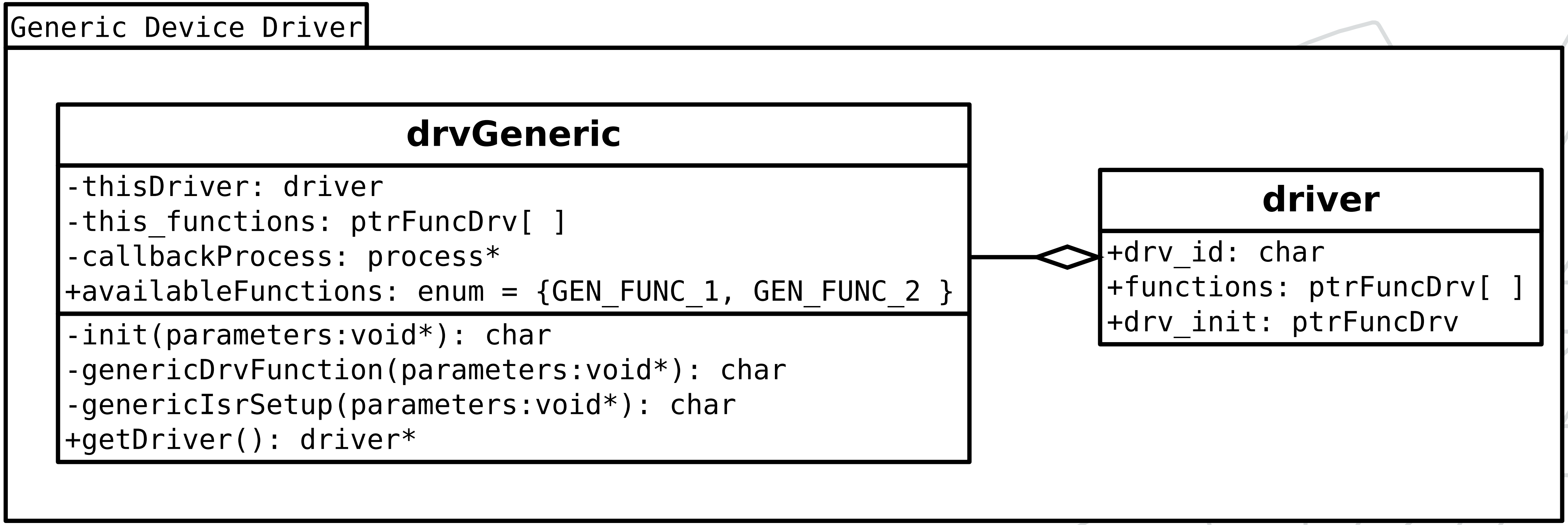
```c
//ptr. de func. para uma função do driver
typedef char(*ptrFuncDrv)(void *parameters);

//estrutura do driver
typedef struct {
    char id;
    ptrFuncDrv *funcoes;
    ptrFuncDrv initFunc;
} driver;
```

driver
+drv_id: char +functions: ptrFuncDrv[ ] +drv_init: ptrFuncDrv

```c
#ifndef DD_TYPES_H
  #define DD_TYPES_H
  //Device Drivers Types (dd_types.h)
  //ptr. de func. para uma função do driver
  typedef char(*ptrFuncDrv)(void *parameters);

  //estrutura do driver
  typedef struct {
    char id;
    ptrFuncDrv *funcoes;
    ptrFuncDrv initFunc;
  } driver;

  //função de retorno do driver
  typedef driver* (*ptrGetDrv)(void);

#endif /* DD_TYPES_H */
```

The standard

Driver example



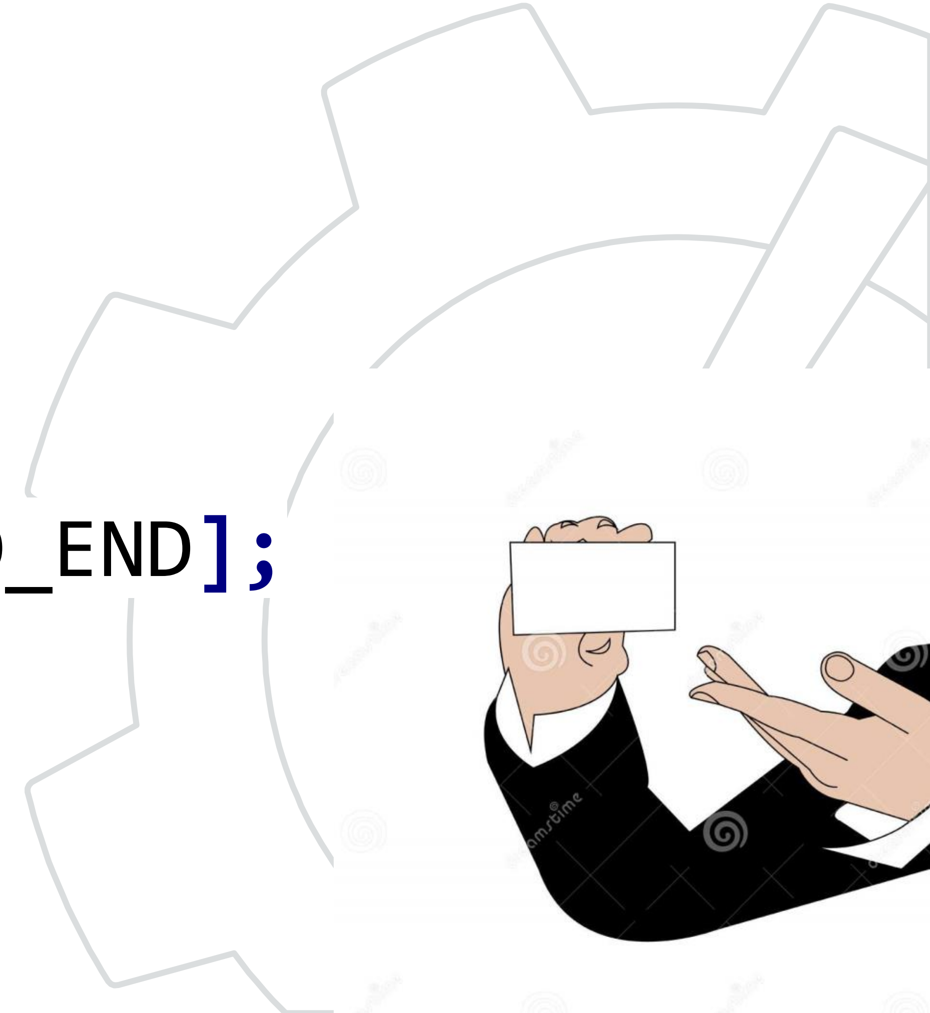
+ visible
- invisible

Driver example

```c
#ifndef drvGeneric_h
    #define drvGeneric_h
    #include "dd_types.h"

    //lista de funções do driver
    enum {
        LED_SET,LED_FLIP, LED_END
    };

    //função de acesso ao driver
    driver* getGenericDriver(void);

#endif // drvGenerico_h
```

Driver example

```c
#include "kernel.h"
#include "pic18f4520.h"
#include "drvGeneric.h“

static driver meu_cartao;

static ptrFuncDrv my_funcs[LED_END];
```



Driver example

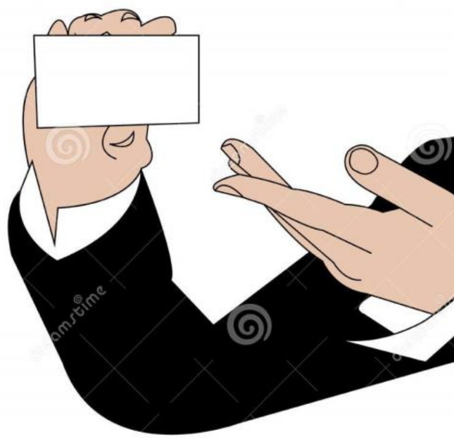
```c
char changePORTD(void *parameters) {
    PORTD = (char) parameters;
    return SUCCESS;
}

char inverte(void * parameters){
    PORTD = ~PORTD;
    return SUCCESS;
}

char initGenerico(void *parameters) {
    TRISD = 0x00; PORTD = 0xFF;
    meu_cartao.id = (char) parameters;
    return SUCCESS;
}
```

Driver example

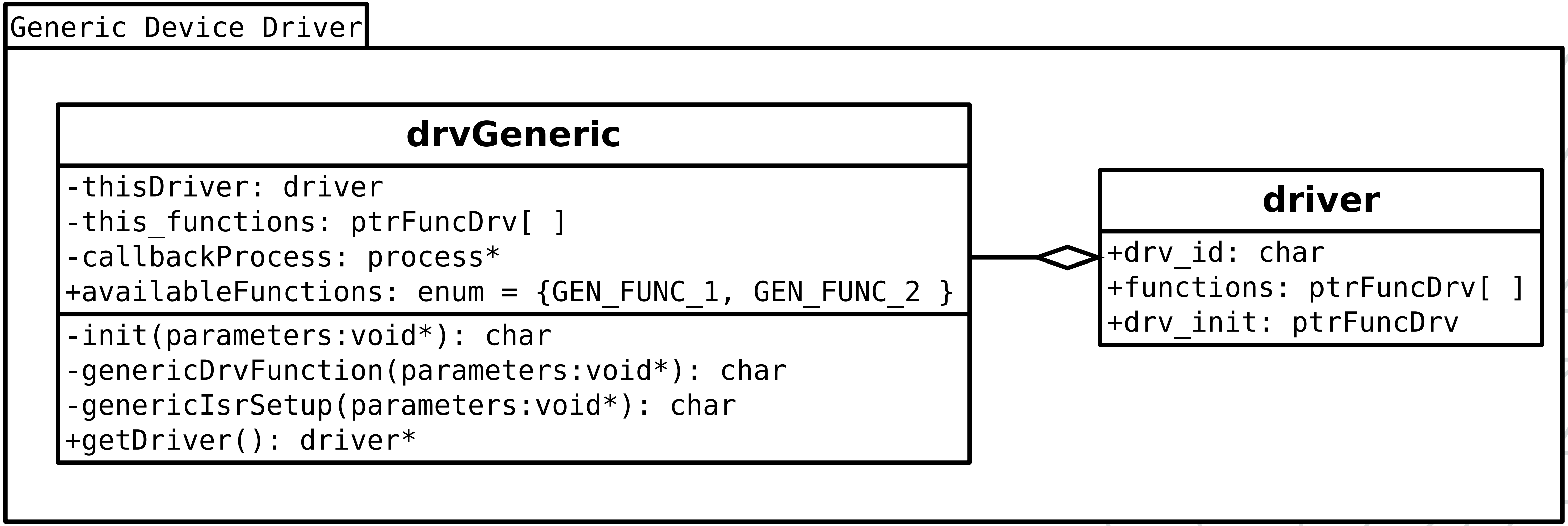
```c
#ifndef drvGeneric_h
    #define drvGeneric_h
    #include "dd_types.h"

    //lista de funções do driver
    enum {
        LED_SET,LED_FLIP, LED_END
    };

    //função de acesso ao driver
    driver* getGenericDriver(void);

#endif // drvGenerico_h
```

Driver example

```
driver* getGenericDriver(void) {
    meu_cartao.initFunc = initGenerico;
    my_funcs[LED_SET] = changePORTD;
    my_funcs[LED_FLIP] = inverte;
    meu_cartao.funcoes = my_funcs;
    return &meu_cartao;
}
```



Driver example

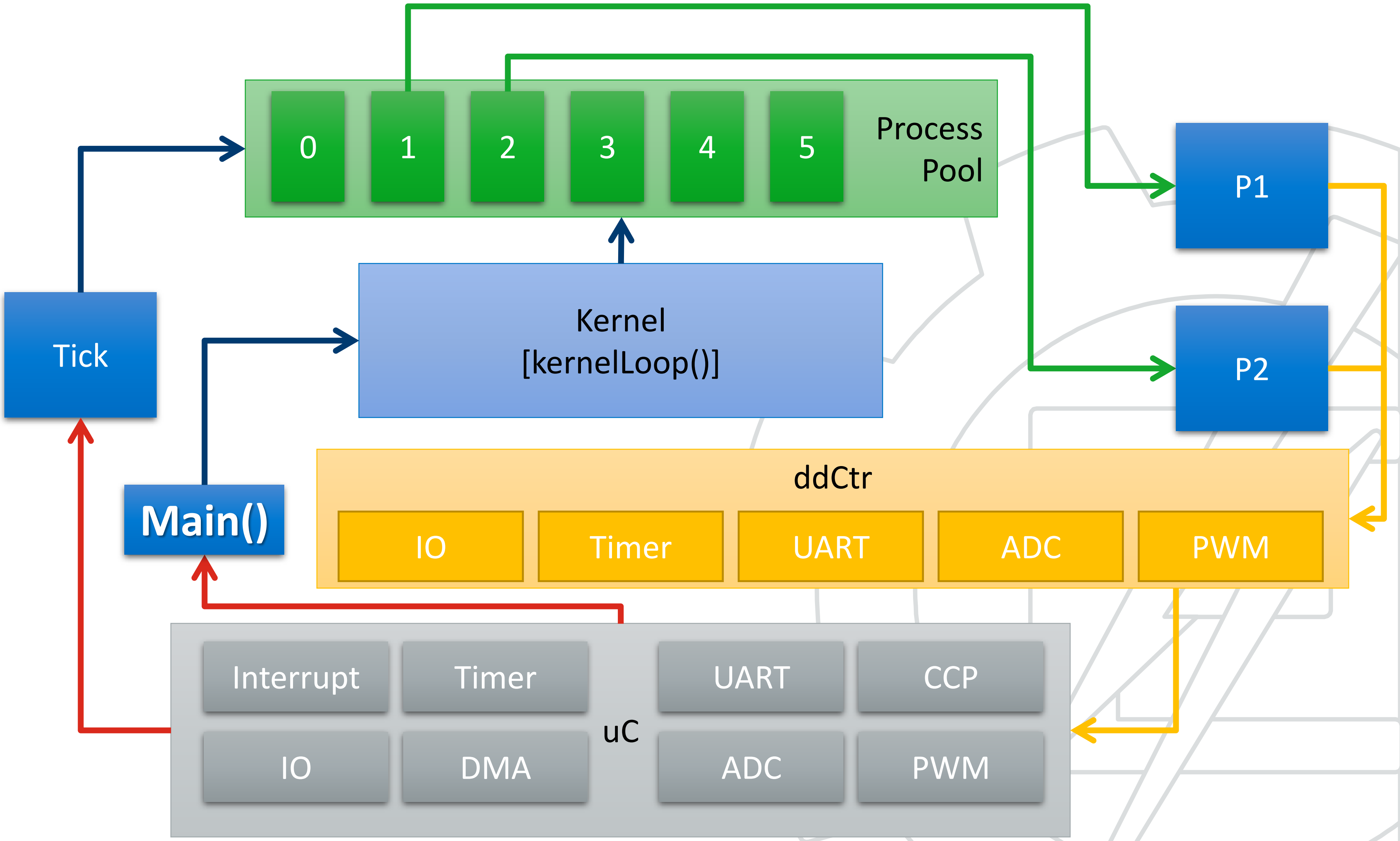


+ visible
- invisible

Device Driver

Controller process

Architecture

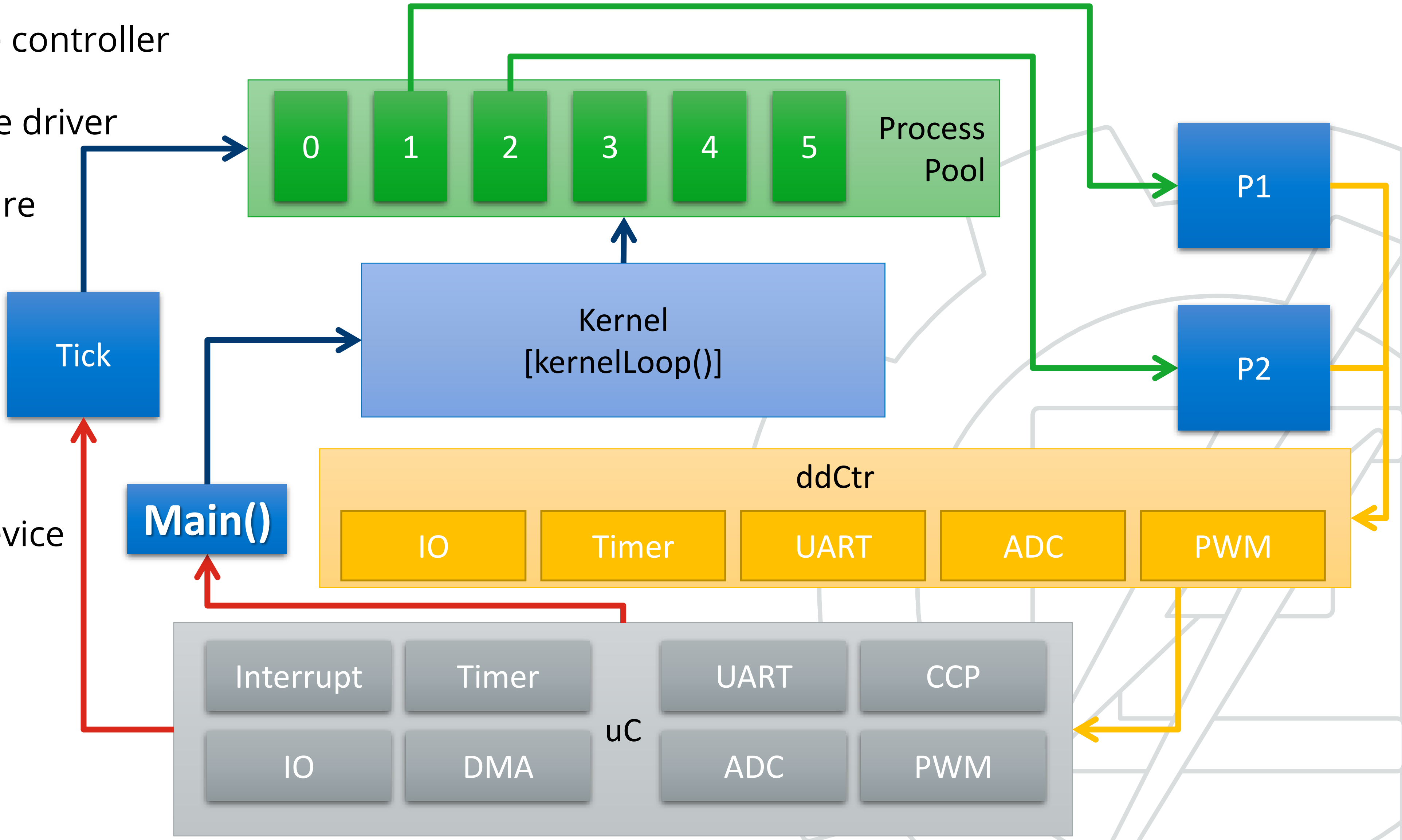


Architecture

1. A Process is created
2. The Process is scheduled
3. The Process access the controller
4. ddController access the driver
5. Driver controls hardware

versus

Direct access to the I/O device



Device Driver Controller

- Used as an <u>interface layer</u> between the kernel and the drivers
- Can "discover" all available drivers (statically or dynamically)
- Store information about <u>all loaded drivers</u>
- Responsible to <u>interpret the messages</u> received from the kernel

Device Driver Controller

- The controller implementation has 2 functions:
  - One for initializing a driver
    - In addition to initialization must provide an ID for the driver.
  - One to go through the application commands for the drivers

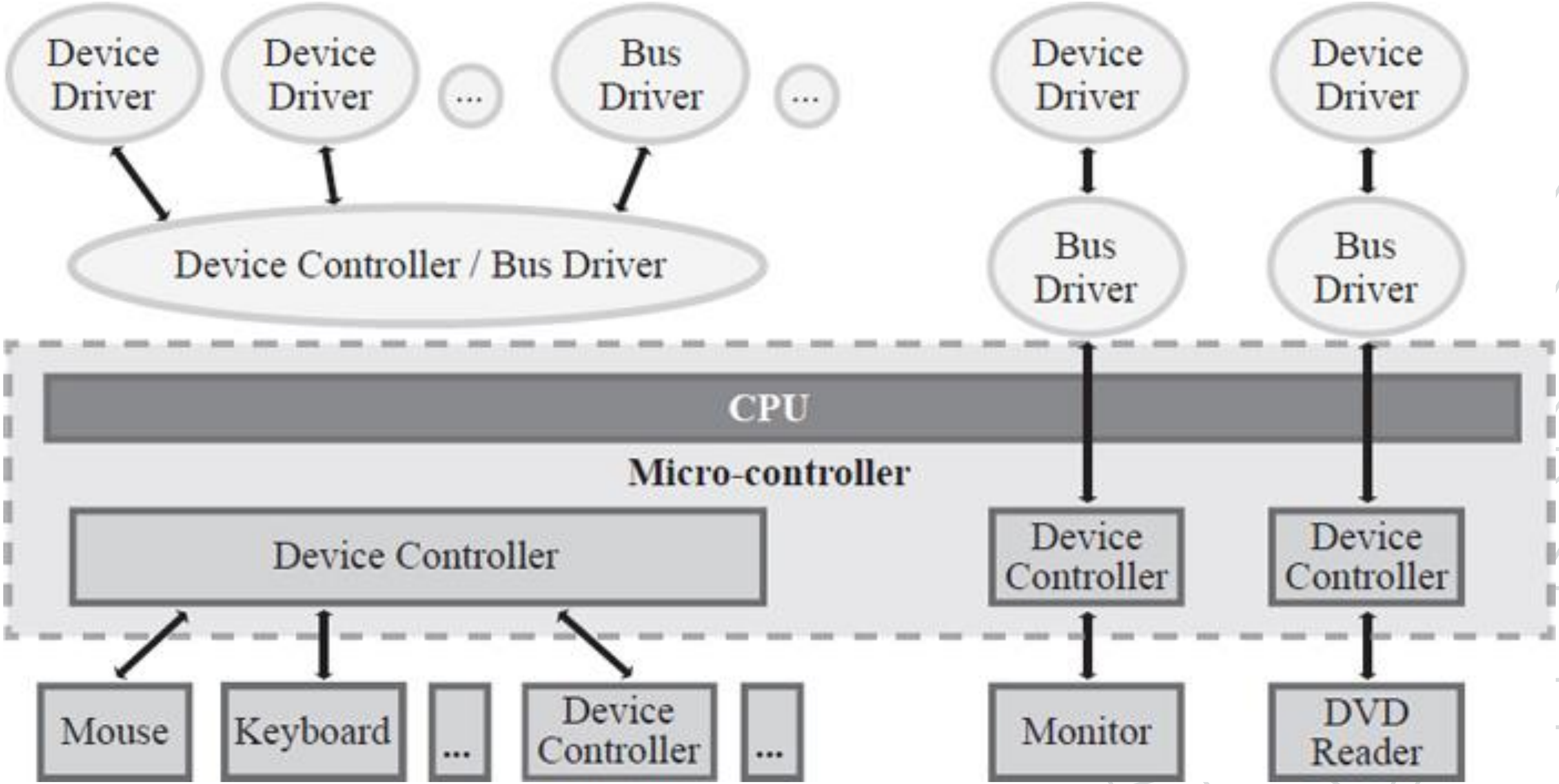
Device Driver Controller

```c
char callDriver(char drv_id, char func_id, void *p) {
  char i;
  for (i = 0; i < dLoaded; i++) {
    //find the right driver
    if (drv_id == drivers[i]->id) {
      return drivers[i]->funcoes[func_id](p);

    }
  }
  return DRV_FUNC_NOT_FOUND; //reliability
}
```

Device Driver Controller

Process and code

Device Driver Controller



Device Driver Controller

```c
static driver* drivers[QNTD_DRV];
static char dLoaded;

char initCtrDrv(void) {
	dLoaded = 0;
}

char initDriver(char newDriver) {
  char resp = FAIL;

  if(dLoaded < QNTD_DRV) {
    //get driver struct
    drivers[dLoaded] = drvGetFunc[newDriver]();

    //should test if driver was loaded correcly: SUCCESS
    resp = drivers[dLoaded]->initFunc(newDriver);
    dLoaded++;
  }
  return resp;
}
```

Device Driver Controller

To gain access to all drivers, it is necessary for the controller to obtain the address of a driver type structure:

- **Statically:**
  - a list is built with functions that return this structure.
- **Dynamically:**
  - this structure is received via serial communication and the address where the data was saved is returned.

Device Driver Controller

Raspberry Pi B
Rev 2 P1 GPIO Header

	Pin No.		
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7

Key

Power +	UART
GND	SPI
I^2C	GPIO

Raspberry Pi B+
B+ J8 GPIO Header

	Pin No.		
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7
DNC	27	28	DNC
GPIO5	29	30	GND
GPIO6	31	32	GPIO12
GPIO13	33	34	GND
GPIO19	35	36	GPIO16
GPIO26	37	38	GPIO20
GND	39	40	GPIO21

Device Driver Controller

Raspberry Pi B
Rev 2 P1 GPIO Header

	Pin No.		
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7

Key

Power +	UART
GND	SPI
I²C	GPIO

Raspberry Pi B+
B+ J8 GPIO Header

	Pin No.		
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7
DNC	27	28	DNC
GPIO5	29	30	GND
GPIO6	31	32	GPIO12
GPIO13	33	34	GND
GPIO19	35	36	GPIO16
GPIO26	37	38	GPIO20
GND	39	40	GPIO21

Device

LCD
printChar (char)
printLine (4x char)
setPosition (2x char)

Device Driver Controller

Raspberry Pi B
Rev 2 P1 GPIO Header

	Pin No.		
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7

Key

Power +	UART
GND	SPI
I²C	GPIO

Raspberry Pi B+
B+ J8 GPIO Header

	Pin No.		
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7
DNC	27	28	DNC
GPIO5	29	30	GND
GPIO6	31	32	GPIO12
GPIO13	33	34	GND
GPIO19	35	36	GPIO16
GPIO26	37	38	GPIO20
GND	39	40	GPIO21

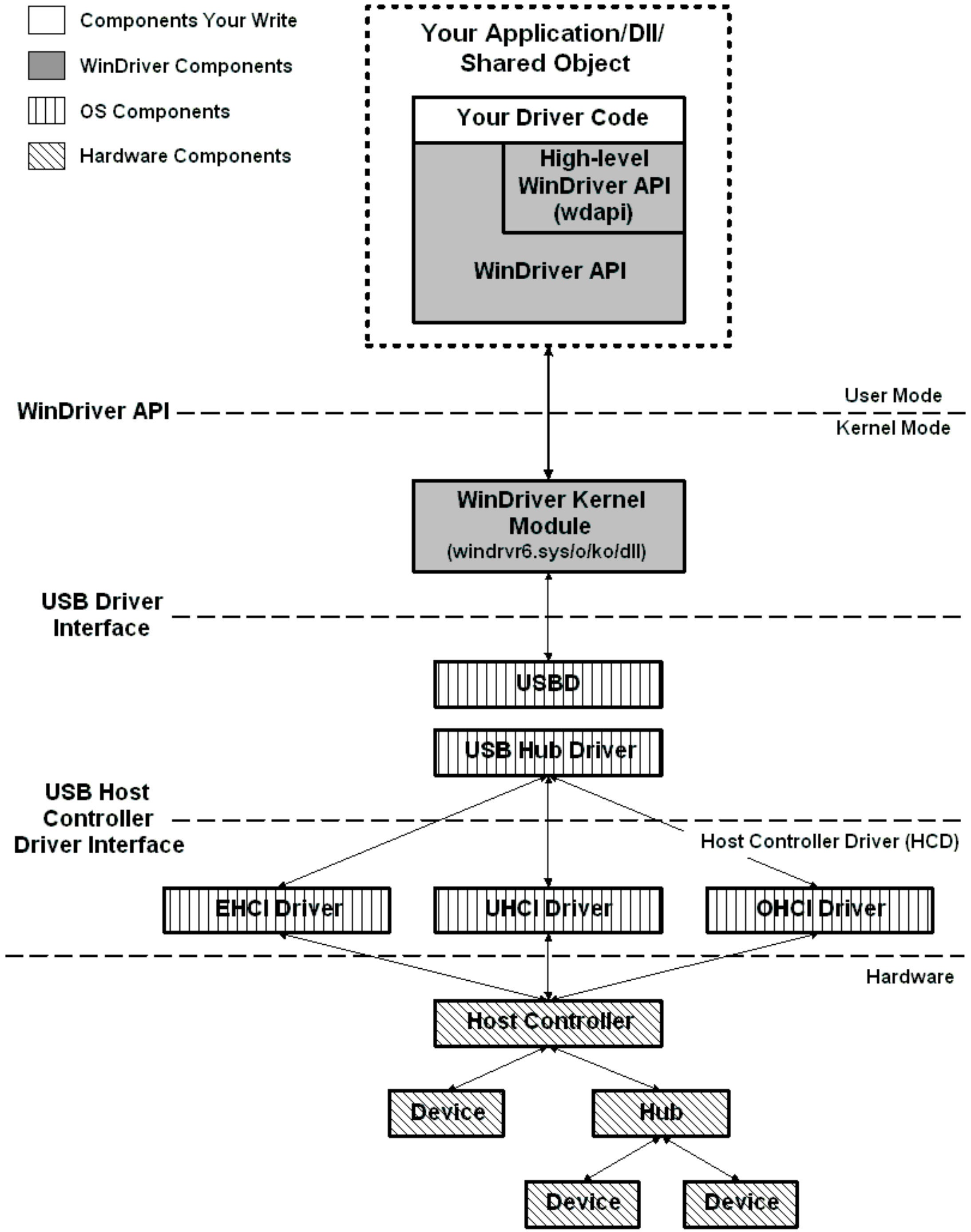
Device

LCD
printChar (char)
printLine (4x char)
setPosition (2x char)

LCD
printChar (void)*
printLine (void)*
setPosition (void)*

Device Driver Controller

Plug&Play and Hot plug



Device Driver Controller

- The drivers functions which return the driver's structure, are presented to the controller in the header file.
  - The controller header includes each header from the drivers it knowns **at run time**.
  - A list of the functions is assembled together with a descriptive enumeration of the same list.

Device Driver Controller

Raspberry Pi B
Rev 2 P1 GPIO Header

	Pin No.		
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7

Key

Power +	UART
GND	SPI
I²C	GPIO

Raspberry Pi B+
B+ J8 GPIO Header

	Pin No.		
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7
DNC	27	28	DNC
GPIO5	29	30	GND
GPIO6	31	32	GPIO12
GPIO13	33	34	GND
GPIO19	35	36	GPIO16
GPIO26	37	38	GPIO20
GND	39	40	GPIO21

Device

LCD
printChar (char)
printLine (4x char)
setPosition (2x char)

LCD
printChar (void)*
printLine (void)*
setPosition (void)*

Device Driver Controller

```c
#ifndef ctrdrv_h
    #define ctrdrv_h
    #define QNTD_DRV 20
    char initCtrDrv(void);
    char callDriver(char drv_id, char func_id, void *parameters);
    char initDriver(char newDriver);

// Static drivers definition
enum {
   DRV_END /*DRV_END must always be the last*/
};

static ptrGetDrv drvGetFunc[DRV_END] = {

};

#endif // ctrdrv_h
```

Device Driver Controller

```c
//ddCtr.h
#include "drvGenerico.h"
#include "drvInterrupt.h"
#include "drvTimer.h"
enum {
    DRV_GEN,                    /*1st driver*/
    DRV_INTERRUPT,              /*2nd driver*/
    DRV_TIMER,                  /*3rd driver*/
    DRV_END /*DRV_END must always be the last*/
};

//the functions to get the drivers should be put in the same order as in the enum
static ptrGetDrv drvGetFunc[DRV_END] = {
    getGenericoDriver,          /*1st driver*/
    getInterruptDriver,         /*2nd driver*/
    getTimerDriver              /*3rd driver*/
};
```

Device driver utilization

```c
void main(void) {
    //system initialization
    kernelInitialization();
    initDriver(DRV_LCD);
    callDriver(DRV_LCD, LCD_CHAR, 'U');
    callDriver(DRV_LCD, LCD_CHAR, 'N');
    callDriver(DRV_LCD, LCD_CHAR, 'I');
    callDriver(DRV_LCD, LCD_CHAR, 'F');
    callDriver(DRV_LCD, LCD_CHAR, 'E');
    callDriver(DRV_LCD, LCD_CHAR, 'I');
    callDriver(DRV_LCD, LCD_CHAR, '@');
    callDriver(DRV_LCD, LCD_CHAR, 'S');
    callDriver(DRV_LCD, LCD_CHAR, '0');
    callDriver(DRV_LCD, LCD_CHAR, '3');
}
```

Where are the defines?

Where are the defines?

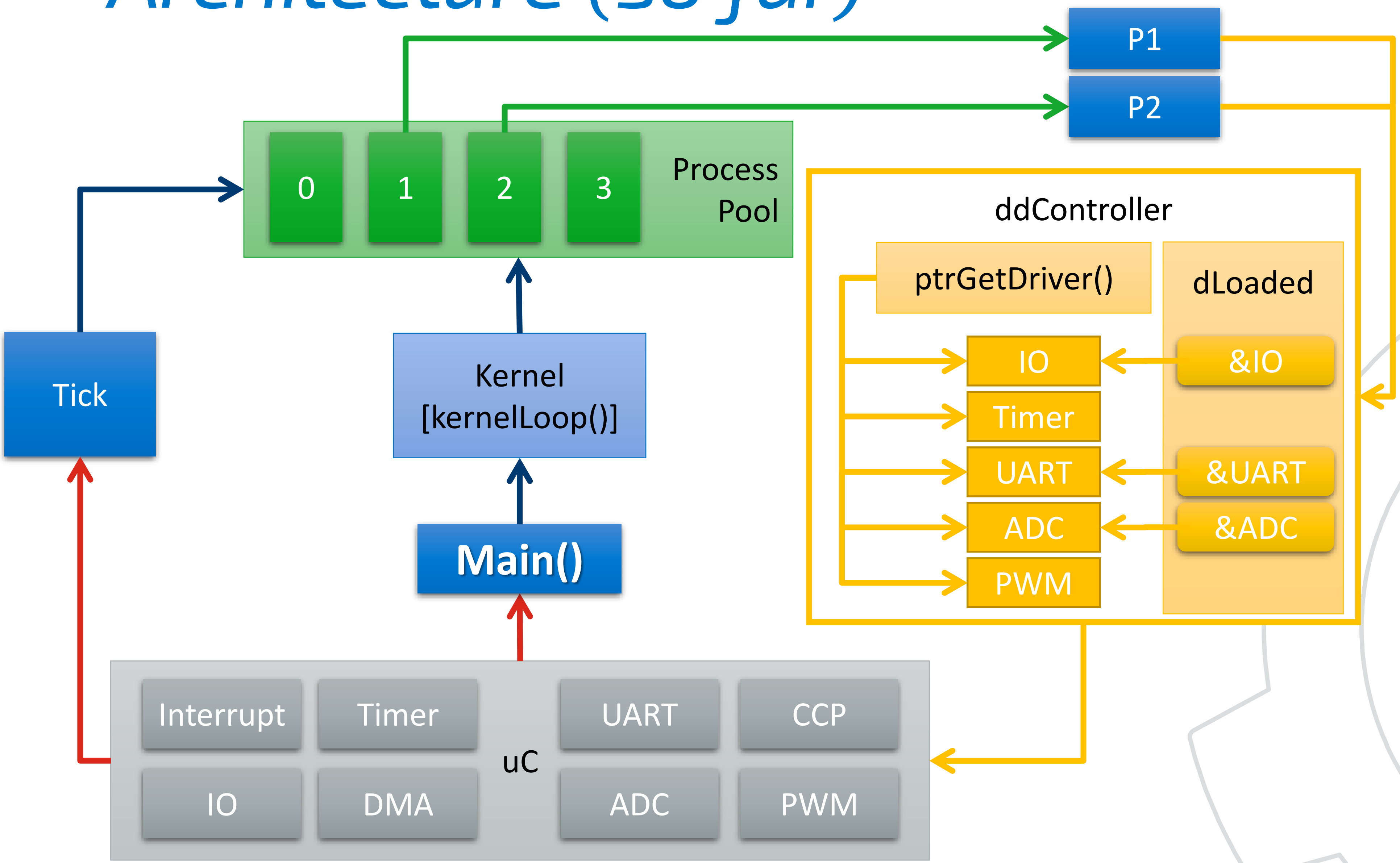
- In order to simplify the design, <u>**each driver**</u> build its function enum

```
enum {
    LCD_COMMAND, LCD_CHAR, LCD_INTEGER, LCD_END
};
```

- <u>**The controller**</u> builds a driver enum

```
enum {
    DRV_INTERRUPT, DRV_TIMER, DRV_LCD, DRV_END
};
```

Architecture (so far)



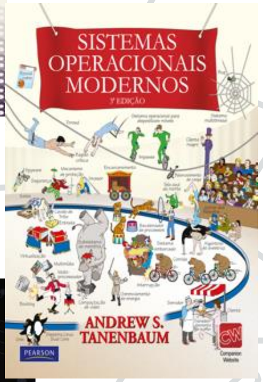
- Pattern that allows kernel integration
- Could define privileged access with control and priority
- Process can't direct access hardware
- MMU can give more security to memory map access
- Could restart only one driver
- ddController block/enable the access to devices (segmentation fault, buffer overflow, malicious injection)

Bibliography

- Denardin, G. B.; Barriquello, C. H. **Sistemas operacionais de tempo real e sua aplicação em sistemas embarcados**. 1ª ed. Editora Blucher. ISBN: 9788521213970. https://plataforma.bvirtual.com.br/Acervo/Publicacao/169968

- Tanenbaum, A.S. **Sistemas Operacionais Modernos**. 3ª ed. 674 páginas. São Paulo: Pearson. ISBN: 9788576052371. Capítulo 5.
  https://plataforma.bvirtual.com.br/Acervo/Publicacao/1233

- Almeida, Moraes, Seraphim e Gomes. **Programação de Sistemas Embarcados**. 2ª ed. Editora GEN LTC. ISBN: 9788595159105.
  https://cengagebrasil.vitalsource.com/books/9788595159112



Available at: https://unifei.edu.br/ensino/bibliotecas/

Embedded Operating Systems

Prof. Otávio Gomes

otavio.gomes@unifei.edu.br

in /otavio-gomes