TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

OPERATING-SYSTEM CONSTRUCTION

Material based on slides by Olaf Spinczyk, Universität Osnabrück

Exercise 2: C++ (2), Keyboard, Interrupts

https://tud.de/inf/os/studium/vorlesungen/betriebssystembau

HORST SCHIRMEIER

Overview

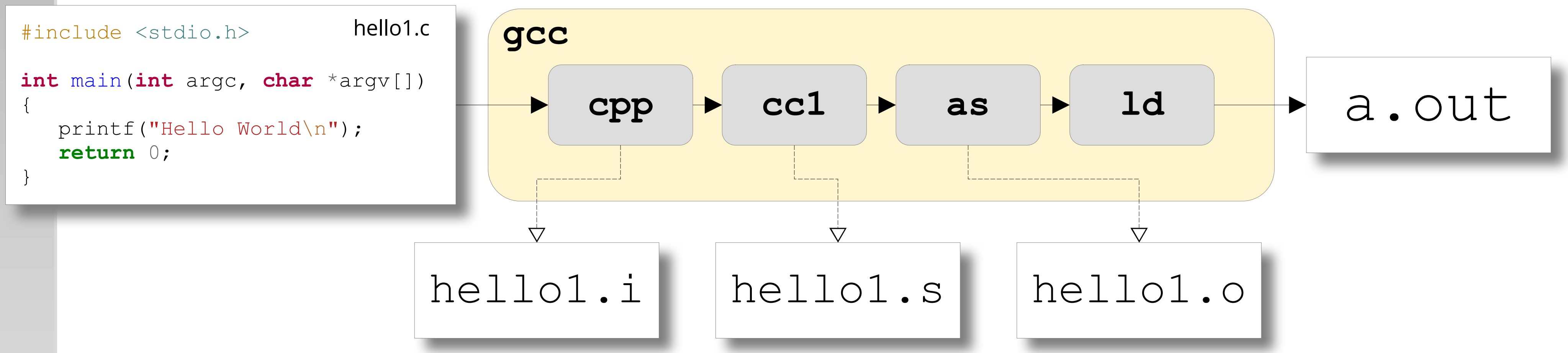
- **C++ Crash Course (Part 2)**
- Lab Task #1: Keyboard
- Interrupts on x86: PIC
- Lab Task #2: Interrupt Handling

More C++ Concepts (Crash Course Part 2)

- Compiling and Linking
- Preprocessor
- Inheritance and Multiple Inheritance
- Virtual Functions

C/C++ Build Process

- Preprocessing, compilation, assembly and linkage in one step: **`gcc hello1.c`**
  - Generates file **`a.out`**
    (name can be changed with parameter `-o`)



hello1.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello World\n");
    return 0;
}
```

C without macros (`gcc -E`)

Assembler code (`gcc -S`)

Machine code: Object file (`gcc -c`)

Source Code – Preprocessor

- Two file extensions:
  - .cc — C++ source code
  - .h — „Header Files“ with definitions of data types, constants, preprocessor macros etc.
- File extensions are only convention, variants:
  - .C, .cpp, .cxx, .hpp, .hh
- The preprocessor textually “integrates” header files in .cc files
  - #include directive:
    - #include <iostream> for system headers
    - #include "device.h" for own header files
  - Modern alternative: C++20 modules

Source Code – Preprocessor

- More preprocessor functionality:
  - Macros, e.g. for constants (*without* semicolon!)

```
#define pi 3.1415926
#define VGA_BASE 0xb8000
```

  - Conditional compilation:

```
#ifdef DEBUG
…
#endif
```

```
#ifndef VGA_BASE
#define VGA_BASE 0xb8000
#endif
```

- The preprocessor **expands macros**, integrates **header-file contents**, and generates a **new text file** (.i) as compiler input.

Source Code – Preprocessor

- Important use-case for #define and #ifndef:
  - Header files may include other headers → infinite recursion possible!
  - Preventing repeated inclusion of header files (“include guard”):

```
#ifndef __cgastr_include__
#define __cgastr_include__

#include "object/o_stream.h"
#include "machine/cgascr.h"

class CGA_Stream
/* Add your code here */
{
/* Add your code here */
};
#endif
```

  - Non-standard alternative: put **`#pragma once`** in the header file

Source Code – Compiler

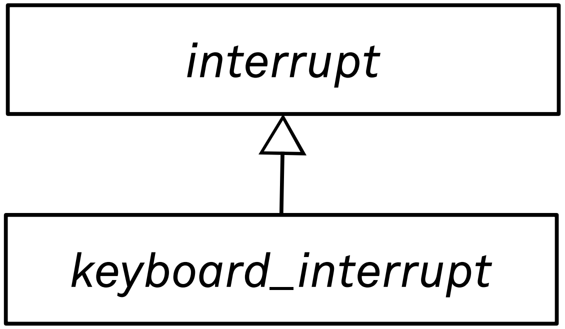
- Generates an object file (.o) from preprocessed source code
  - Generally **not directly executable**: unresolved references to functions or variables from other object files
- Checks code for syntactic and semantic correctness, may
  - … abort compilation and print an **error** message *(errors)*
  - … emit **warnings** that could be a sign of a problem
  - Warnings do not abort compilation, but do not ignore them!

Source Code – Linker

- Links a set of object files (.o) and possibly libraries (.a, .so) to an executable binary:
  - Resolve references
  - Sort/group object-file parts/sections in memory map of executable
- Two linking modes:
  - **dynamic:** Libraries are loaded when starting the program, reference resolution at start- or even at runtime (“lazy linkage”)
  - **static:** Libraries are linked at link/build time, yielding a completely linked “static” binary containing all external dependencies.

Single Inheritance

- Class *keyboard_interrupt* inherits from class *interrupt*
- Inheritance operator “:” (like “extends” in Java)



interrupt.h:

```
class interrupt {
   …
};
```

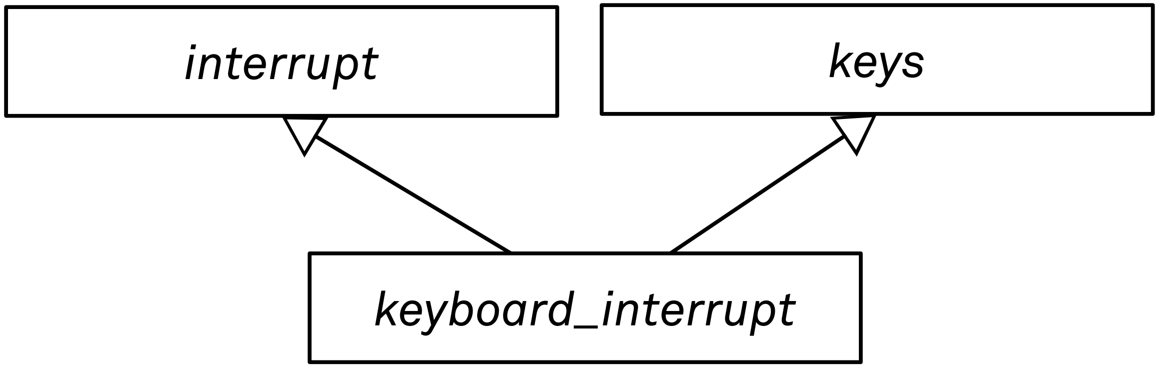
keyboard_interrupt.h:

```
#include "interrupt.h"

class keyboard_interrupt : public interrupt {
public:
  keyboard_interrupt();
  ~keyboard_interrupt();
};
```

Multiple Inheritance

- Class *keyboard_interrupt* inherits from classes *interrupt* **and** *keys*:



interrupt

keys

keyboard_interrupt

keyboard_interrupt.h:

```
#include "interrupt.h"

class keyboard_interrupt : public interrupt, public keys {
public:
  keyboard_interrupt();
  ~keyboard_interrupt();
};
```

Virtual Functions

- Specially “marked” function of a base class (keyword: `virtual`)
- Derived class may **override** it, thereby providing a **specialized implementation** for its instances (however, this also works with non-virtual functions)
- For classes with ≥1 virtual functions, each object “knows” from which class in the hierarchy it was instantiated → correct function gets called in polymorphic scenarios
- **Not every function is virtual** by default (unlike in Java)

Virtual Functions

- Output:

  ”Derived”

- without **virtual** in front of void base::display():

  ”Base”

```
#include <iostream>

class base {
public:
  virtual void display() {
    cout << "Base";
  }
};

class derived : public base {
public:
  void display() {
    cout << "Derived";
  }
};

void main() {
  base *ptr = new derived;
  ptr->display();
}
```

Virtual Destructors

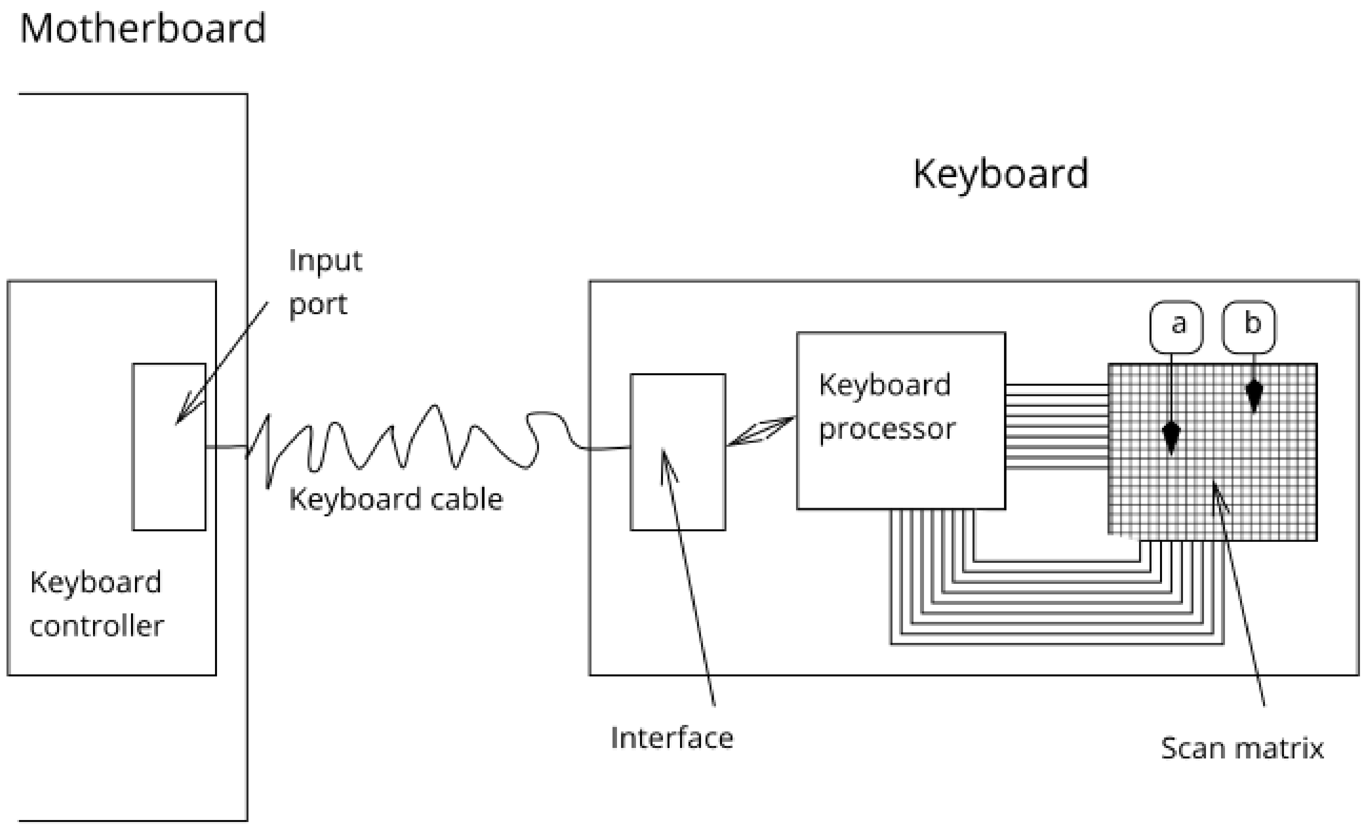
- **Rule of thumb:** A class with a **virtual function** should also have a **virtual destructor**
  - A **non-virtual destructor** does not guarantee correct destruction of derived classes.
    (If one exists anyways, this can even be interpreted such that its author didn't intend (and doesn't recommend) deriving from this class.)

Overview

- C++ Crash Course (Part 2)
- **Lab Task #1: Keyboard**
- Interrupts on x86: PIC
- Lab Task #2: Interrupt Handling

PC Keyboard

- classic:

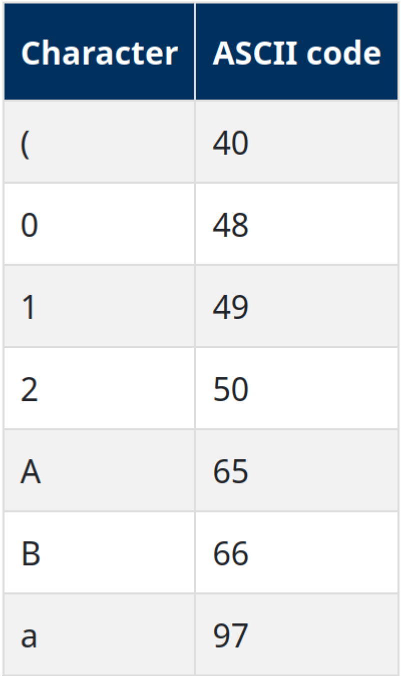


- modern PC: USB keyboard
  - *USB Legacy Support*: Programming still also works via keyboard controller (backwards compatibility)

Key Encoding

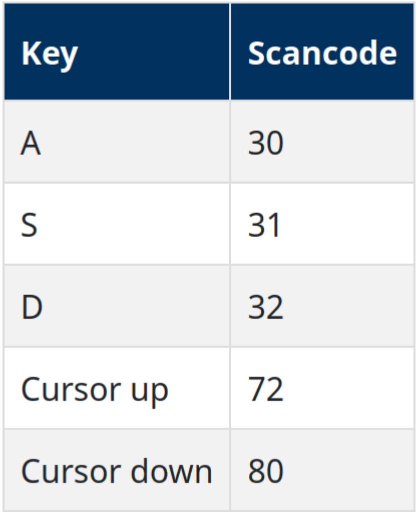
- Each key has unique code (“Scan code”)
  - 7-bit number (max. 128 keys)

Representation in applications (and in CGA video memory!): Character codes (ASCII)



Character	ASCII code
(	40
0	48
1	49
2	50
A	65
B	66
a	97

Representation in keyboard hardware: Key codes



Key	Scancode
A	30
S	31
D	32
Cursor up	72
Cursor down	80

- Keyboard sends additional information
  - *Make Code* when pressing / while holding a key
  - *Break Code* when releasing a key

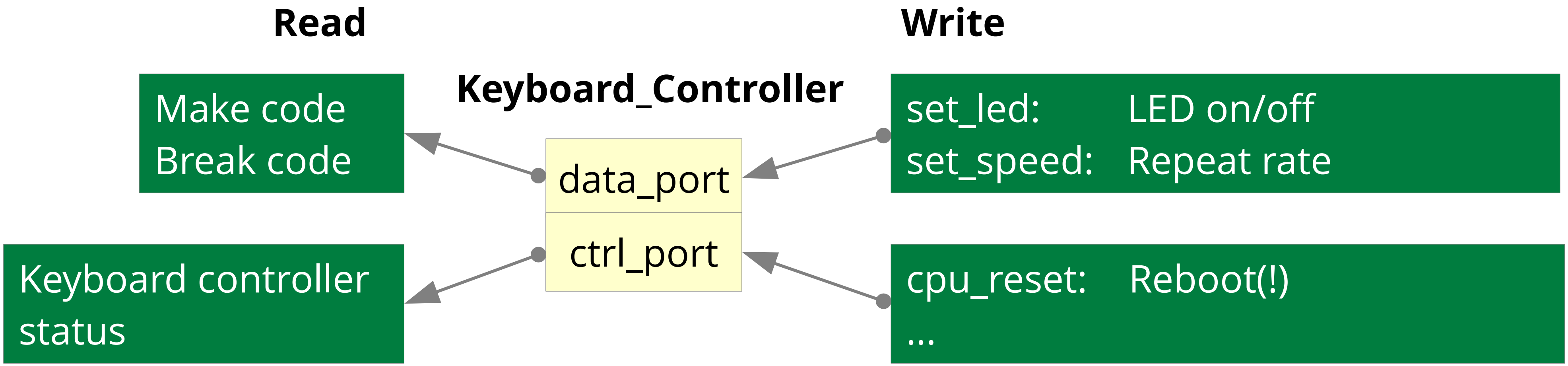
Make and Break Codes

- General system:
  - Make code (key pressed) = Scan code
  - Break code (key released) = Scan code + 128 (Bit 7)
- Some keys send **more than one code**
  - e.g. function keys (F1–F12)
  - … for historic reasons (XT keyboard)
  - up to 3 make/break codes per key
- Built-in **repeat** functionality
  - Hardware sends additional make codes while holding a key

→ Decoding is cumbersome
  - already implemented in OOStuBS template: **bool** *key_decoded*()

Communication with Keyboard

- Keyboard controller: two I/O ports
  - Input/output register (data_port) 0x60
  - Control register (ctrl_port) 0x64



Read

Write

Keyboard_Controller

data_port

ctrl_port

Make code
Break code

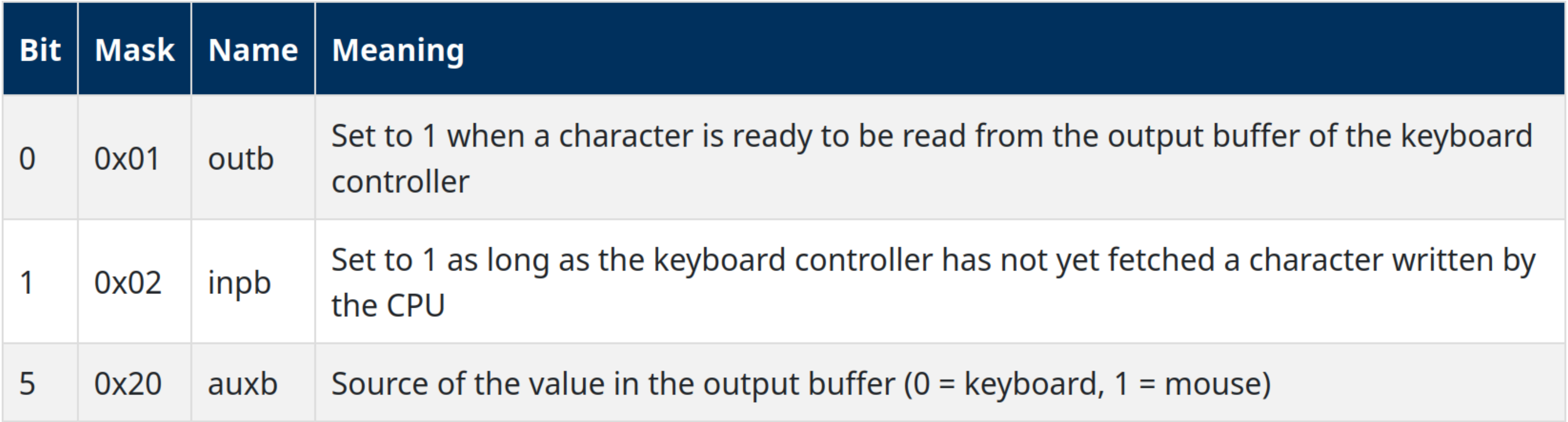
Keyboard controller status

set_led: LED on/off
set_speed: Repeat rate

cpu_reset: Reboot(!)
…

Keyboard-Controller Status

- Status register:



Bit	Mask	Name	Meaning
0	0x01	outb	Set to 1 when a character is ready to be read from the output buffer of the keyboard controller
1	0x02	inpb	Set to 1 as long as the keyboard controller has not yet fetched a character written by the CPU
5	0x20	auxb	Source of the value in the output buffer (0 = keyboard, 1 = mouse)

Keyboard-Controller Status – Usage

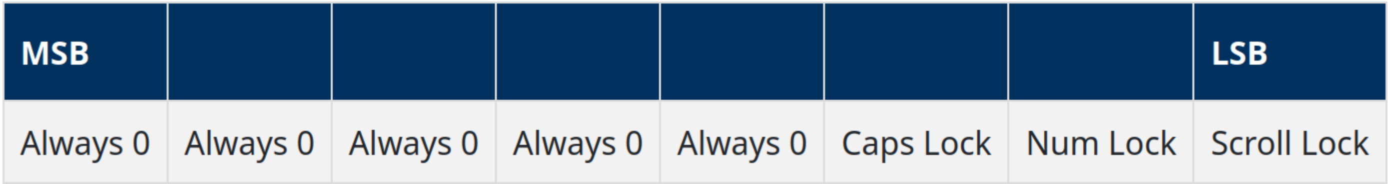
Bit	Mask	Name	Meaning
0	0x01	outb	Set to 1 when a character is ready to be read from the output buffer of the keyboard controller
1	0x02	inpb	Set to 1 as long as the keyboard controller has not yet fetched a character written by the CPU
5	0x20	auxb	Source of the value in the output buffer (0 = keyboard, 1 = mouse)

- Active keyboard polling (without interrupts):
  - Wait until **outb** in ctrl_port is set (1)
  - Read *Make/Break* code from data_port (clears ctrl_port.**outb**)
- Program keyboard (set_led, set_speed)
  - Write **command byte** to data_port
  - Keyboard replies with **ACK** (0xfa), need to wait for this reply (see above)
  - Write **data byte** to data_port (LED codes, repeat rate)
  - Keyboard replies with **ACK**, need to wait for this reply

Keyboard Programming

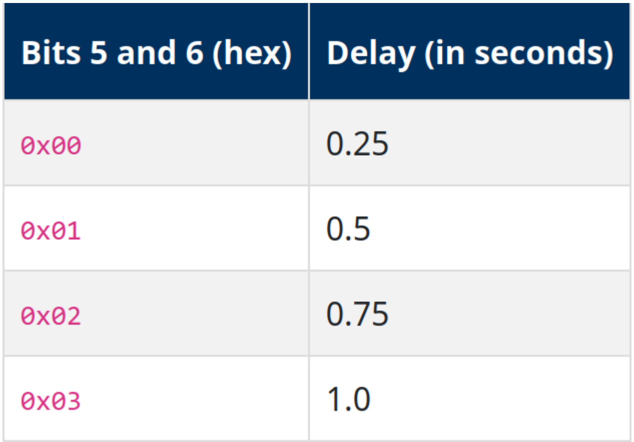
- set_led **0xed**, <led_mask> in data_port
- set_speed **0xf3**, <config_byte> in data_port

Parameter for **set_led** command: (led_mask)

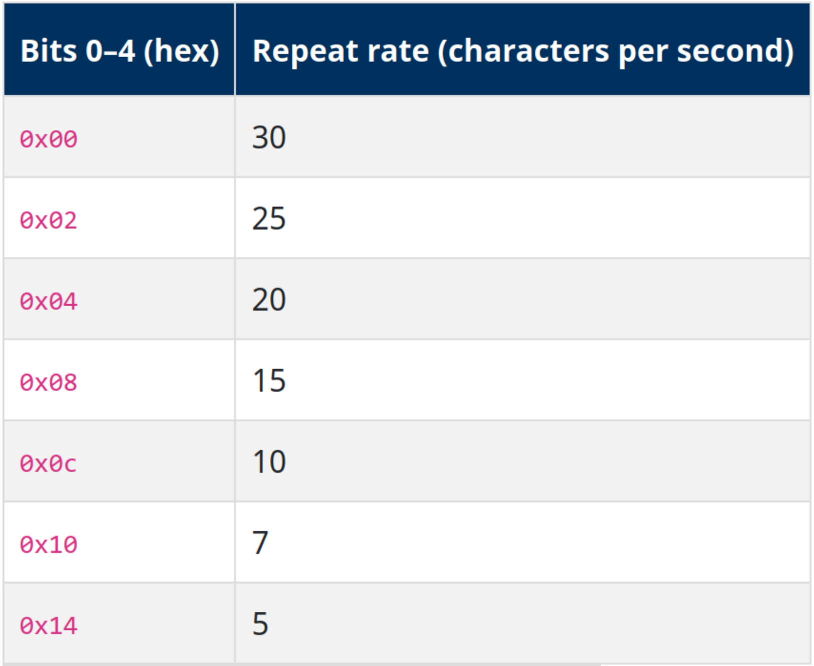


MSB							LSB
Always 0	Always 0	Always 0	Always 0	Always 0	Caps Lock	Num Lock	Scroll Lock

Parameter for **set_speed** command: (config_byte)



Bits 5 and 6 (hex)	Delay (in seconds)
0x00	0.25
0x01	0.5
0x02	0.75
0x03	1.0



Bits 0-4 (hex)	Repeat rate (characters per second)
0x00	30
0x02	25
0x04	20
0x08	15
0x0c	10
0x10	7
0x14	5

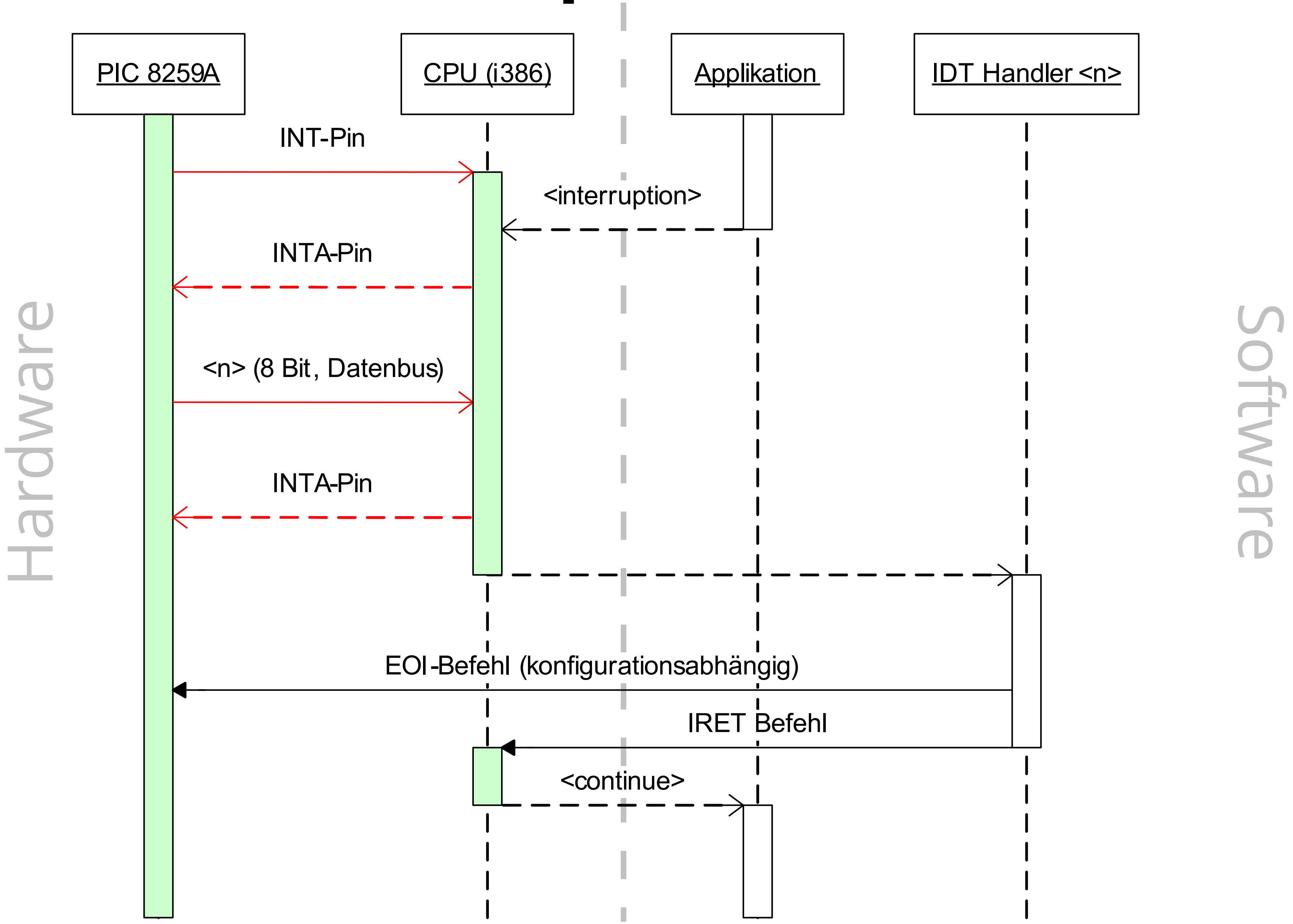
Overview

- C++ Crash Course (Part 2)
- Lab Task #1: Keyboard
- **Interrupts on x86: PIC**
- Lab Task #2: Interrupt Handling

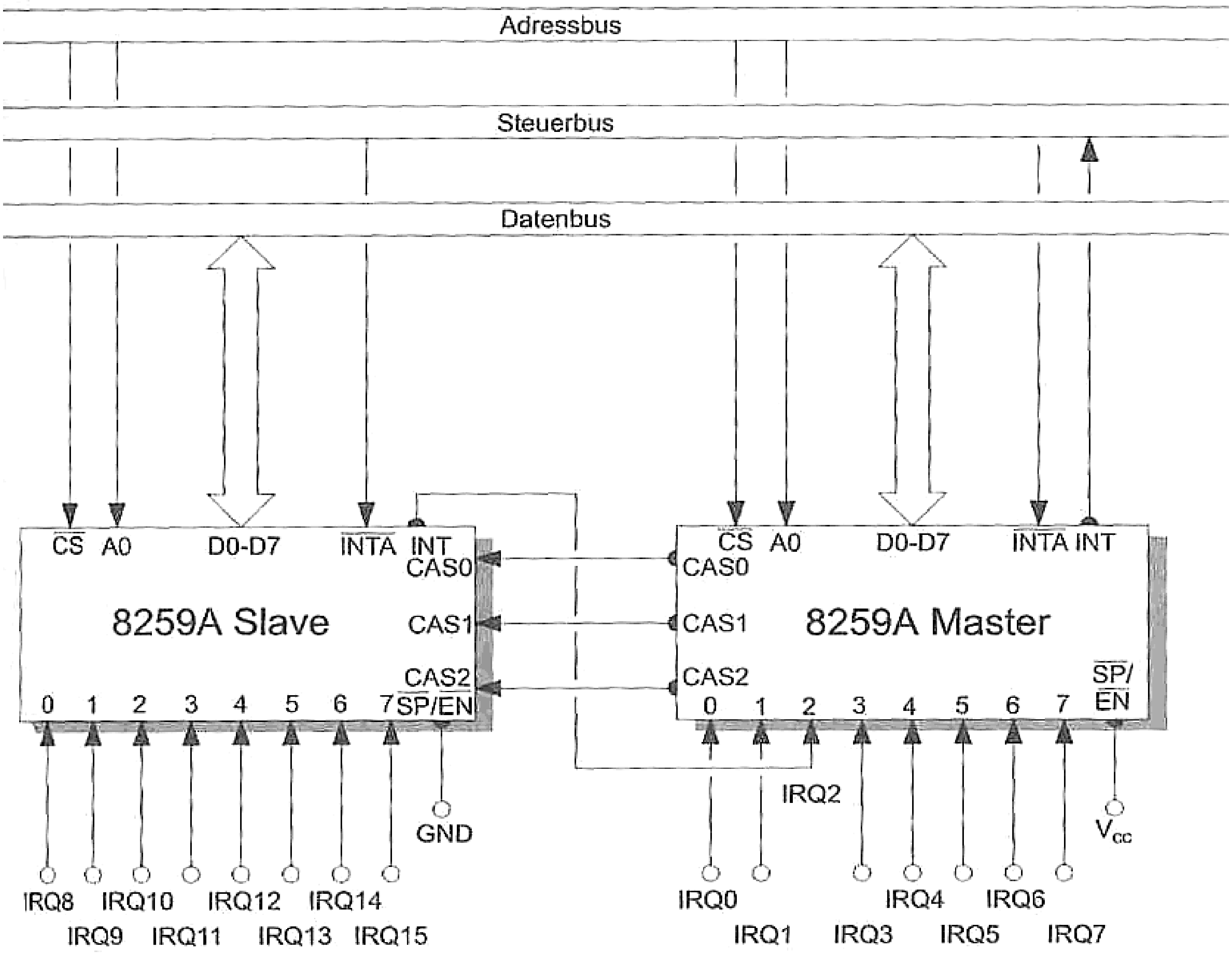
Hardware IRQs on x86 CPUs

- x86 CPUs up to and including i486:
  only one interrupt line (INT) + one NMI line
  - INT can be masked with IE bit in EFLAGS register
    - `cli` instruction (clear interrupt enable flag) – **disable interrupt handling**
    - `sti` instruction (set interrupt enable flag) – **enable interrupt handling**
  - NMI cannot be masked in the CPU (“non-maskable interrupt”)
    - … PC still allows this via CPU-external hardware …
- External controller puts IRQ number on memory bus
  - PC: **Programmable Interrupt Controller** (PIC) 8259A
  - Communication protocol between CPU and PIC 8259A

Hardware IRQ Sequence (with PIC)

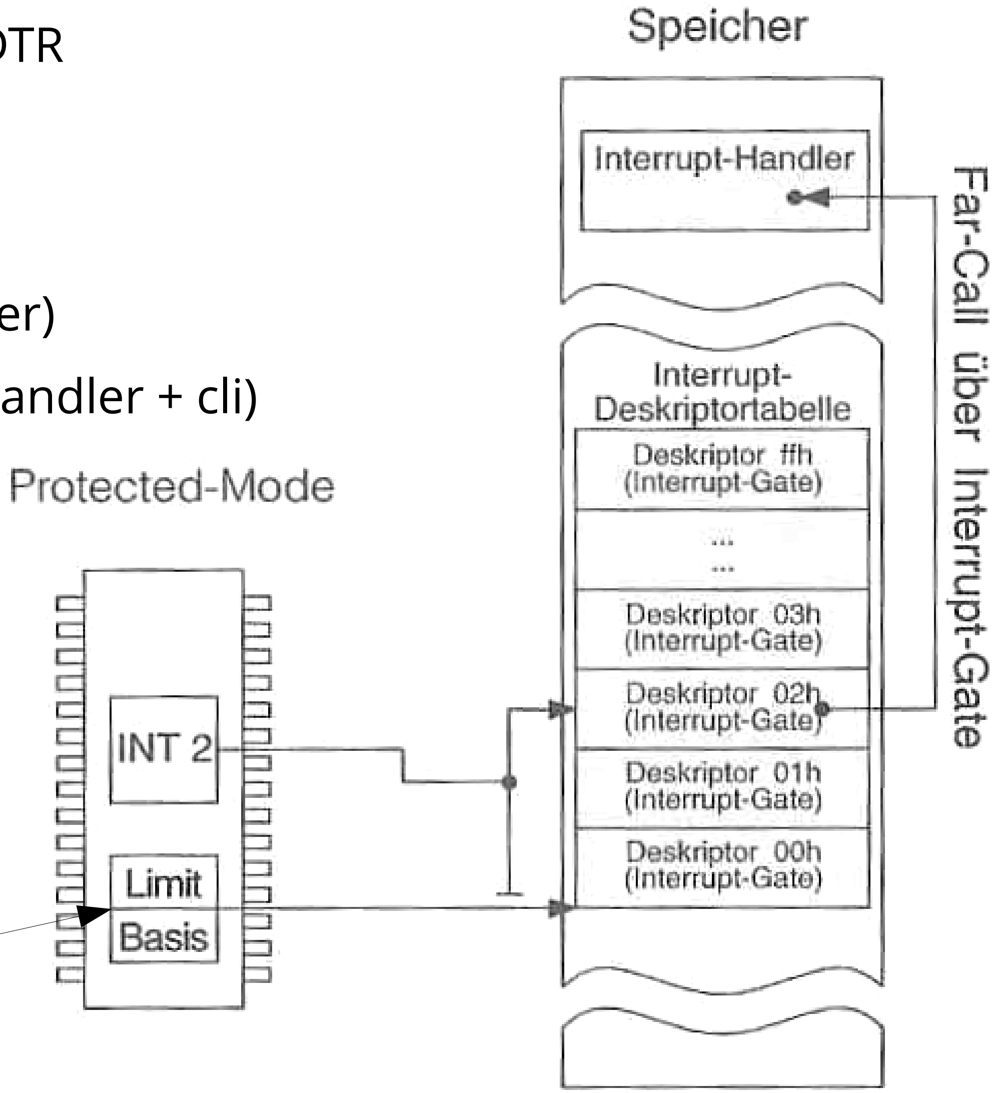


PIC Cascading in the PC (15 Interrupts)



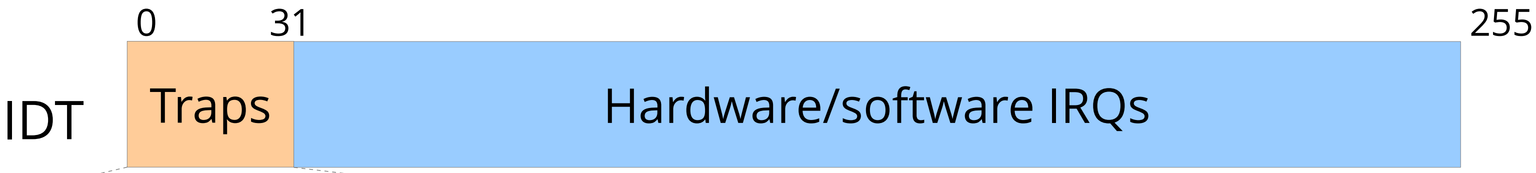
x86-64 Interrupt Descriptor Table

- max. 256 entries
  - Base address and size in IDTR
- 16 bytes per entry (“gate”)
  - Task gate (Hardware tasks)
  - Trap gate (Exception handler)
  - Interrupt gate (Exception handler + cli)


Speicher
Interrupt-Handler
Far-Call über Interrupt-Gate
Interrupt-Deskriptortabelle
Deskriptor ffh (Interrupt-Gate)
...
...
Deskriptor 03h (Interrupt-Gate)
Deskriptor 02h (Interrupt-Gate)
Deskriptor 01h (Interrupt-Gate)
Deskriptor 00h (Interrupt-Gate)
Protected-Mode
INT 2
Limit
Basis

IDTR

x86 IDT: Structure

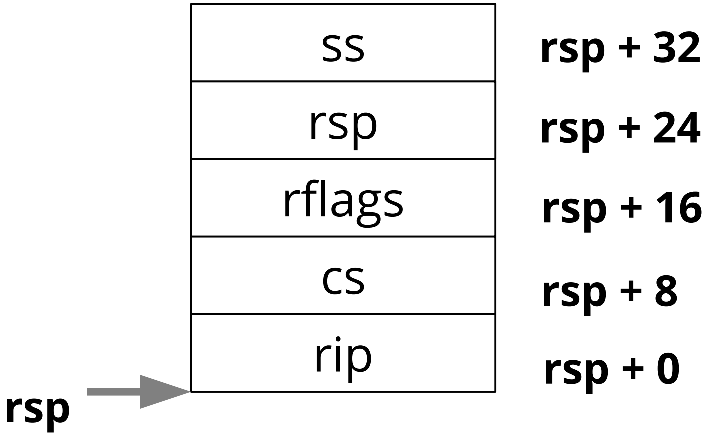


Number	Description
0	**Divide-by-zero**
1	Debug exception
2	Non-Maskable Interrupt (NMI)
3	**Breakpoint (INT 3)**
4	Overflow
5	Bound exception
6	**Invalid Opcode**
7	FPU not available
8	Double Fault
9	Coprocessor Segment Overrun
10	Invalid TSS
11	Segment not present
12	Stack exception
13	**General Protection**
14	Page fault
15	Reserved
16	**Floating-point error**
17	Alignment Check
18	Machine Check
19-31	Reserved By Intel

- Entries 0–31 for traps (fixed)
- Trap = Exception that occurs synchronously to control flow
  - Division by 0
  - Page fault
  - Breakpoint
  - …
- Entries 32–255 for IRQs (configurable)
  - Software (`INT` <number>)
  - Hardware (CPU's INT pin to HIGH, #number on data bus)

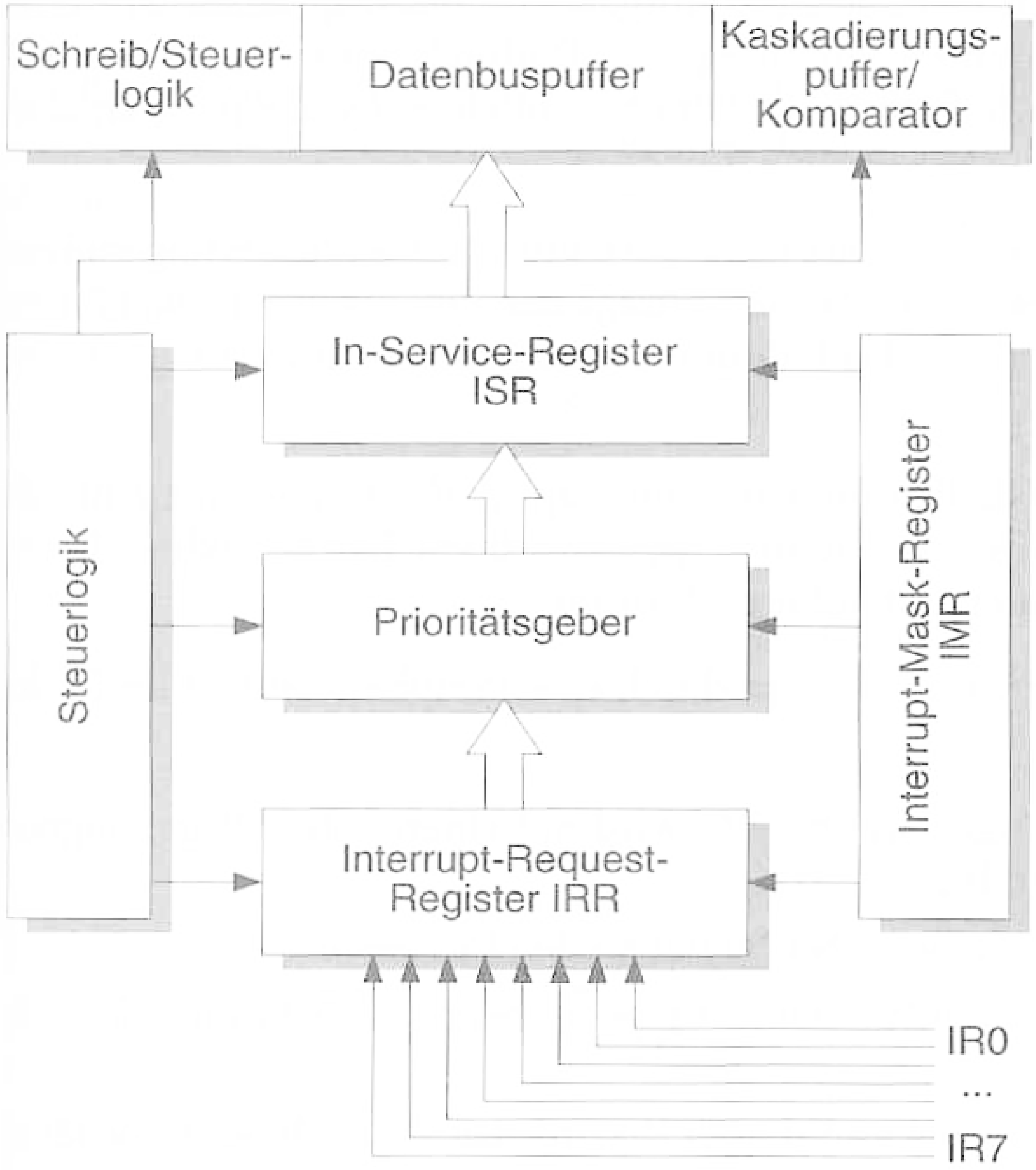
State Saving

- When an interrupt occurs, the CPU **automatically saves a part of its state** on the stack
  - Active stack segment (ss)
  - Stack pointer (rsp)
  - Condition codes (rflags)
  - Active code segment (cs)
  - Return address (rip)
  - For some exceptions (= “traps”): additionally an error code (8 bytes)
- Automatically saved state is restored by `iretq` instruction
  - If handler uses other registers, it **must save/restore them by itself**!



ss	**rsp + 32**
rsp	**rsp + 24**
rflags	**rsp + 16**
cs	**rsp + 8**
rip	**rsp + 0**

PIC 8259A – Internal Structure

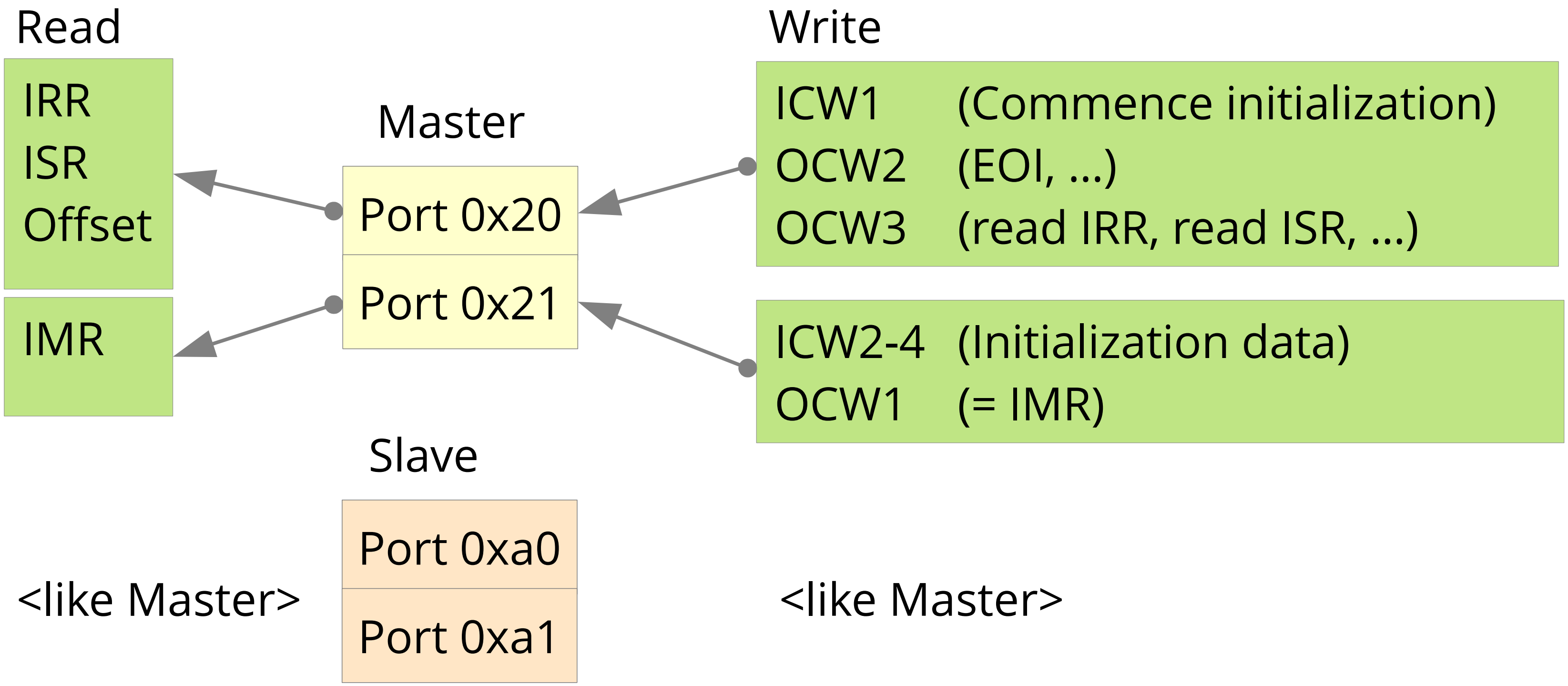


highest

lowest

Accessing PICs via I/O Ports

- Each PIC has 2 ports that can be read/written
- Data that can be written: ICW1–4, OCW1–3
  - ICW = **Initialization Control Word** – PIC initialization
  - OCW = **Operation Control Word** – Commands during operation
- Read data depends on command



Read

IRR
ISR
Offset

IMR

Master

Port 0x20

Port 0x21

Write

ICW1 (Commence initialization)
OCW2 (EOI, …)
OCW3 (read IRR, read ISR, …)

ICW2-4 (Initialization data)
OCW1 (= IMR)

Slave

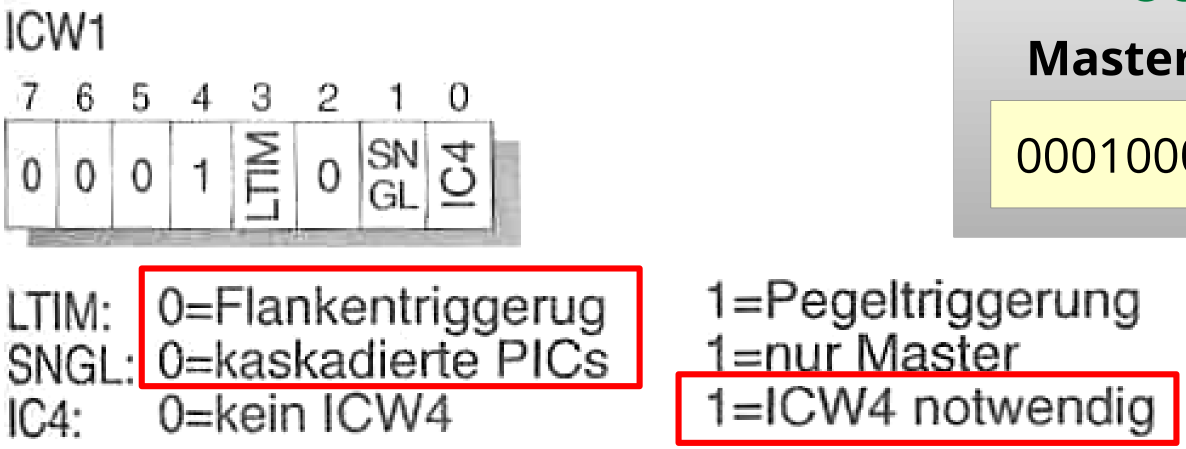
<like Master>

Port 0xa0

Port 0xa1

<like Master>

PIC Initialization – Part 1



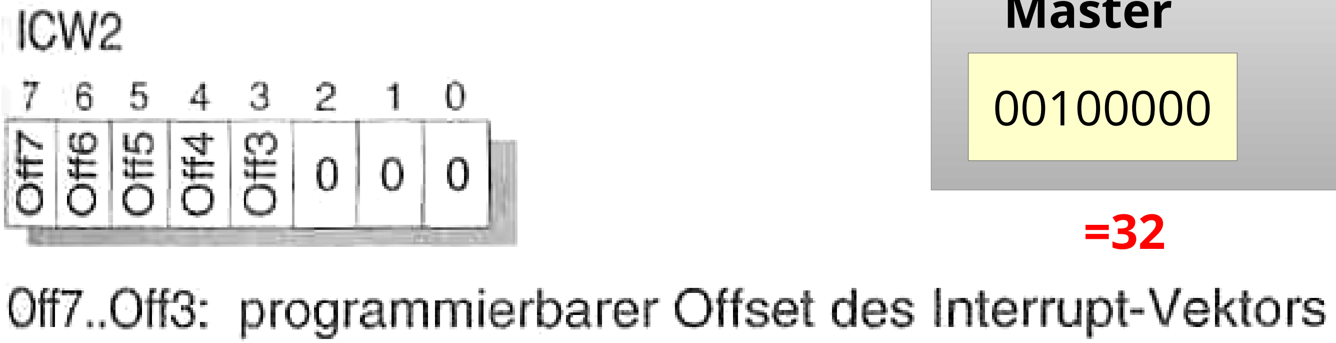
ICW1

7	6	5	4	3	2	1	0
0	0	0	1	LTIM	0	SNGL	IC4

LTIM: 0=Flankentriggerug 1=Pegeltriggerung
SNGL: 0=kaskadierte PICs 1=nur Master
IC4: 0=kein ICW4 1=ICW4 notwendig

OOStuBS setting:

Master	Slave
00010001	00010001



ICW2

7	6	5	4	3	2	1	0
Off7	Off6	Off5	Off4	Off3	0	0	0

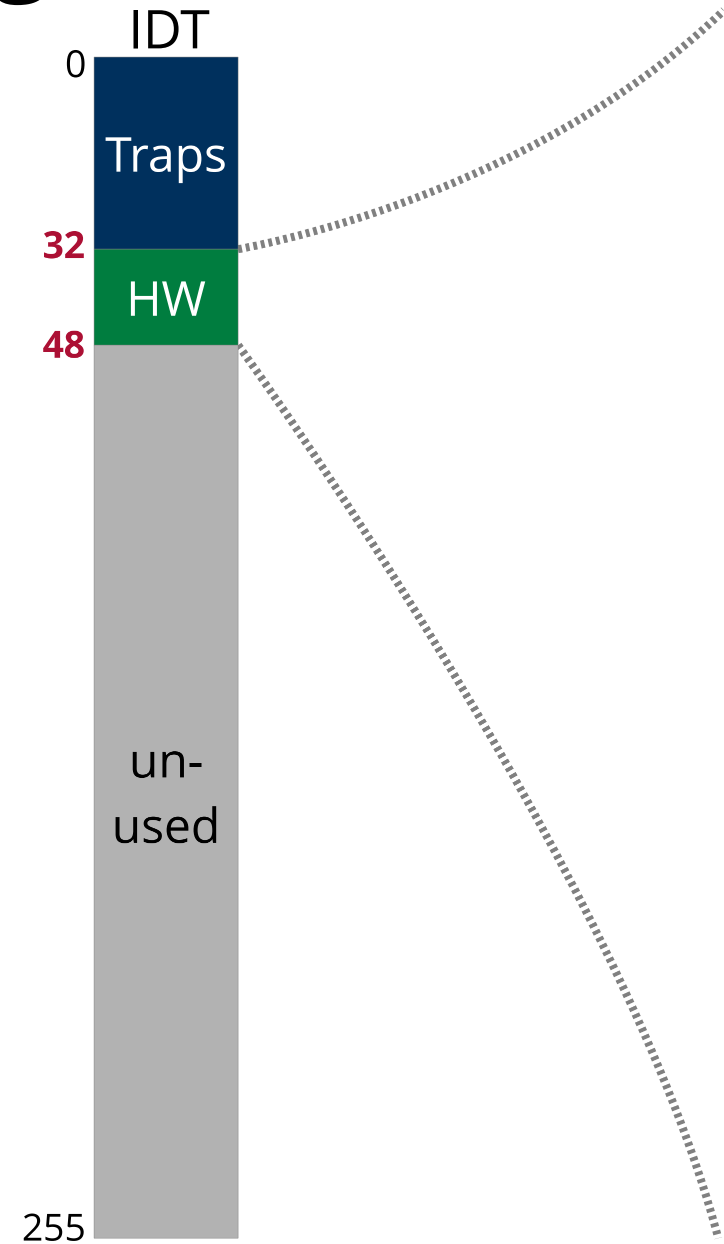
Off7..Off3: programmierbarer Offset des Interrupt-Vektors

OOStuBS setting:

Master	Slave
00100000	00101000
=32	=40

Mapping of HW IRQs (OOStuBS)

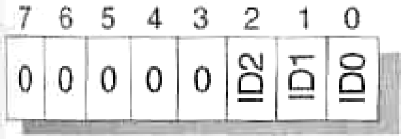
Standard AT IRQ mapping



IRQ	Description
0	Programmable Interrupt Timer (PIT)
1	Keyboard
2	(PIC Cascade)
3	COM2
4	COM1
5	LPT2
6	Floppy-Disk Drive
7	LPT1 / spurious interrupt
8	CMOS Real-Time Clock
9	
10	
11	
12	PS/2 Mouse
13	FPU / Coprocessor / Inter-Processor
14	Primary ATA HDD
15	Secondary ATA HDD

PIC Initialization – Part 2

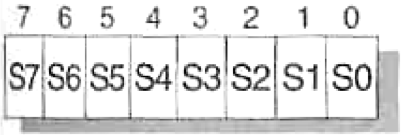
ICW3 (Slave)



7	6	5	4	3	2	1	0
0	0	0	0	0	ID2	ID1	ID0

ID2..ID0: Identifizierungsnummer des Slave-PIC

ICW3 (Master)



7	6	5	4	3	2	1	0
S7	S6	S5	S4	S3	S2	S1	S0

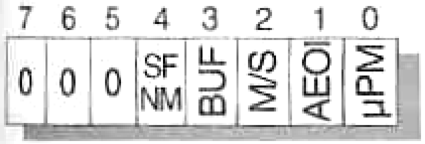
S7..S0: 0=zugehörige IR-Leitung ist mit Peripheriegerät verbunden oder frei
1=zugehörige IR-Leitung ist mit Slave-PIC verbunden

OOStuBS setting:

Master	Slave
00000100	0000010

IRQ 2 → Slave **ID 2**

ICW4



7	6	5	4	3	2	1	0
0	0	0	SF NM	BUF	M/S	AEOI	µPM

OOStuBS setting:

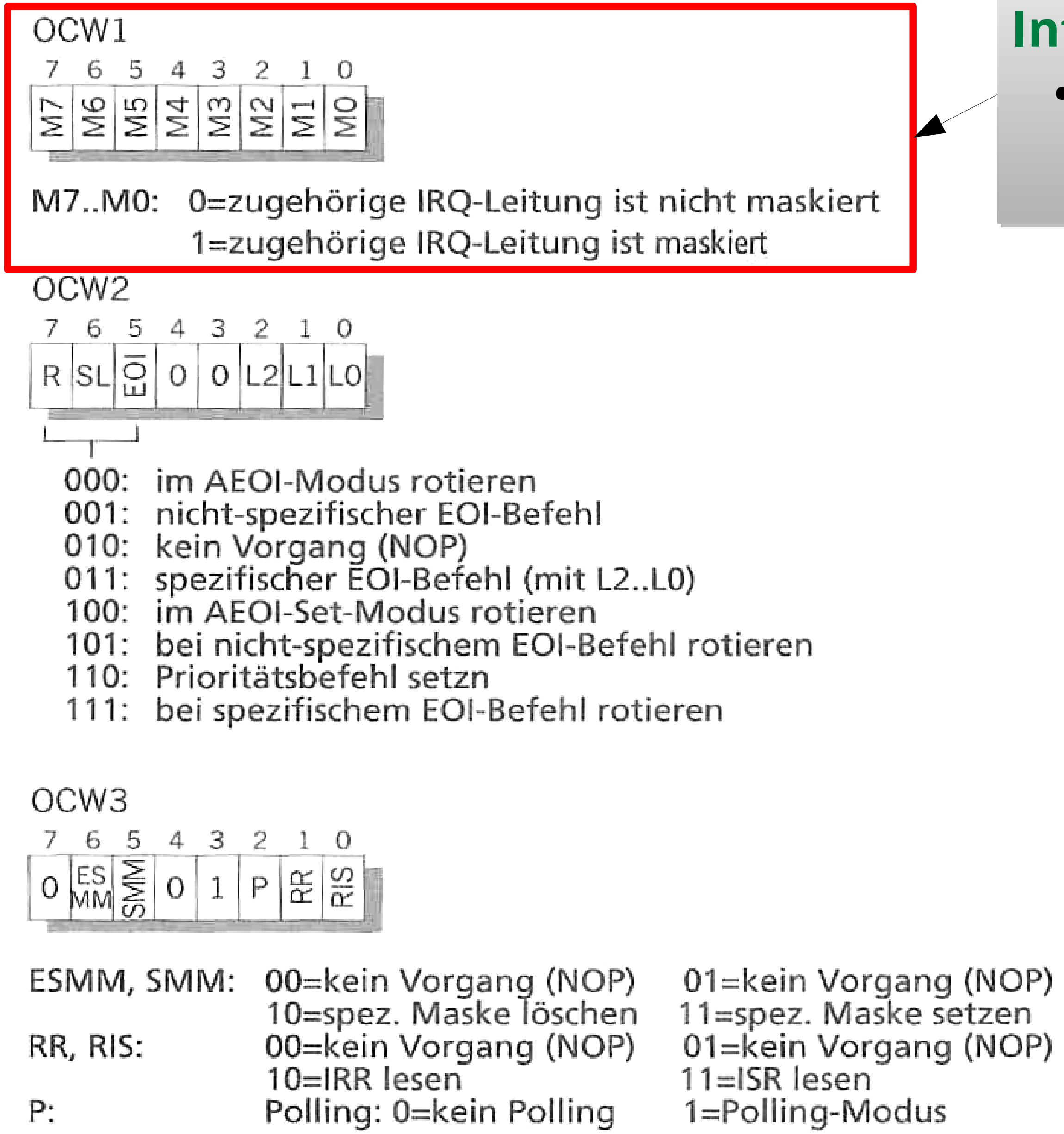
Master	Slave
00000011	00000011

SFNM:	0=kein Special-Fully-Nested-Modus	1=Special-Fully-Nested-Modus
BUF:	0=kein gepufferter Modus	1=gepufferter Modus
M/S:	0=Slave-PIC	1=Master-PIC
AEOI:	0=manueller EOI	1=automatischer EOI
µPM:	0=Betrieb im MCS-80/85-Modus	1=Betrieb im 8086/88-Modus

PIC Programming

Interrupt mask (IMR)

- read and write via Port 0x21 / 0xa1



OCW1

7	6	5	4	3	2	1	0
M7	M6	M5	M4	M3	M2	M1	M0

M7..M0: 0=zugehörige IRQ-Leitung ist nicht maskiert
1=zugehörige IRQ-Leitung ist maskiert

OCW2

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

000:	im AEOI-Modus rotieren
001:	nicht-spezifischer EOI-Befehl
010:	kein Vorgang (NOP)
011:	spezifischer EOI-Befehl (mit L2..L0)
100:	im AEOI-Set-Modus rotieren
101:	bei nicht-spezifischem EOI-Befehl rotieren
110:	Prioritätsbefehl setzn
111:	bei spezifischem EOI-Befehl rotieren

OCW3

7	6	5	4	3	2	1	0
0	ESMM	SMM	0	1	P	RR	RIS

ESMM, SMM:	00=kein Vorgang (NOP)	01=kein Vorgang (NOP)
	10=spez. Maske löschen	11=spez. Maske setzen
RR, RIS:	00=kein Vorgang (NOP)	01=kein Vorgang (NOP)
	10=IRR lesen	11=ISR lesen
P:	Polling: 0=kein Polling	1=Polling-Modus

Overview

- C++ Crash Course (Part 2)
- Lab Task #1: Keyboard
- Interrupts on x86: PIC
- **Lab Task #2: Interrupt Handling**

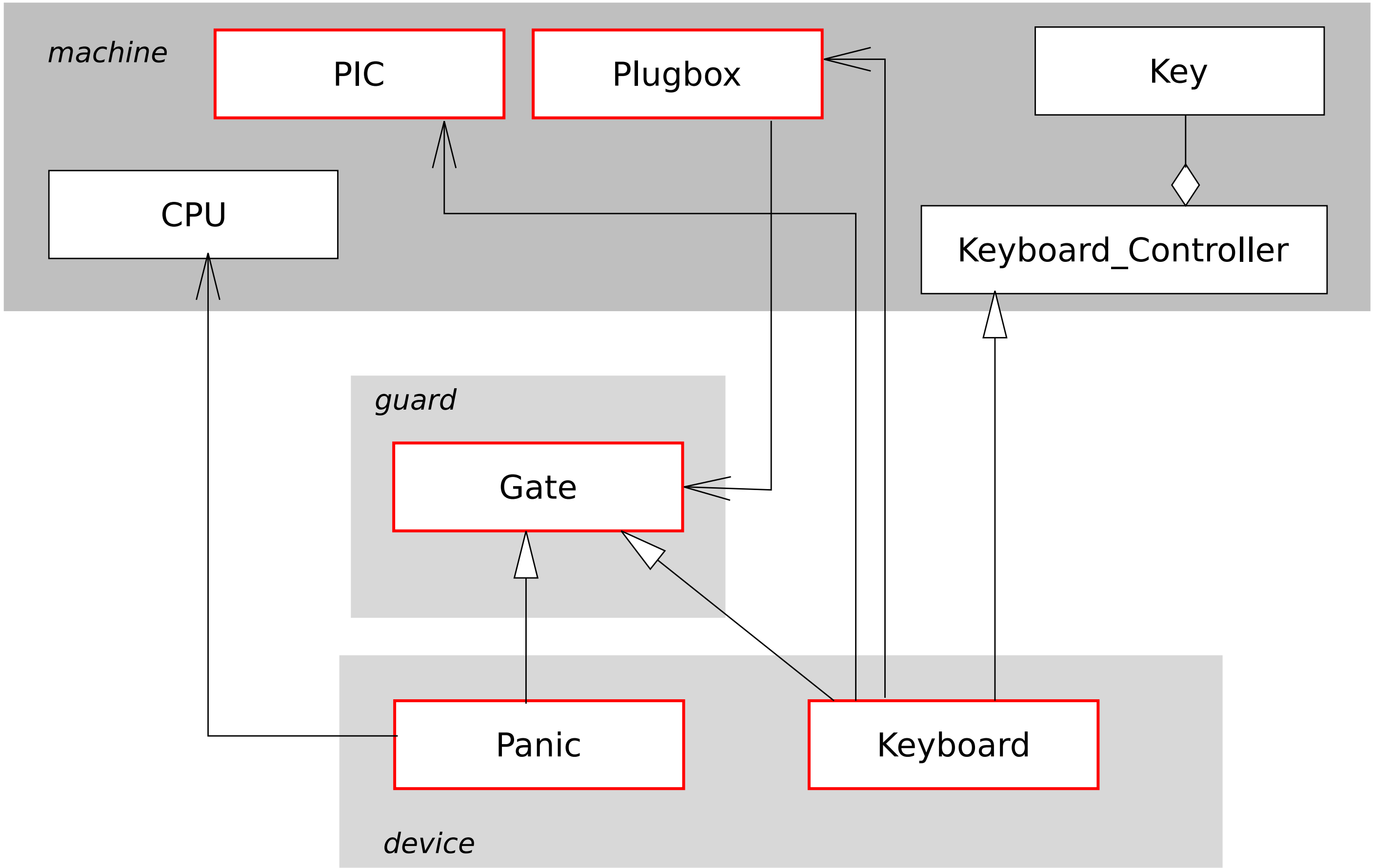
Interrupt Handler in OOStuBS

- Interrupt handling starts in `guardian()` function
  - Parameter `slot`: IRQ number

```
void guardian( unsigned int slot ) {
   ... // call IRQ handler (Gate object)
}
```

  - During interrupt handling, interrupts are disabled
    - Can be manually re-enabled via `sti` (wrapped in `CPU::enable_int()`)
    - Automatically re-enabled when `guardian()` returns (why?)
- Actual (IRQ-specific) IRQ handlers
  - are instances of class Gate
  - are registered/unregistered in class Plugbox

Interrupt Handler in OOStuBS


machine
PIC
Plugbox
Key
CPU
Keyboard_Controller
guard
Gate
Panic
Keyboard
device