Übung zu Betriebssystembau (Ü BS)

Interruptbehandlung in OOStuBS

Wanja Hofer

Lehrstuhl für Informatik IV

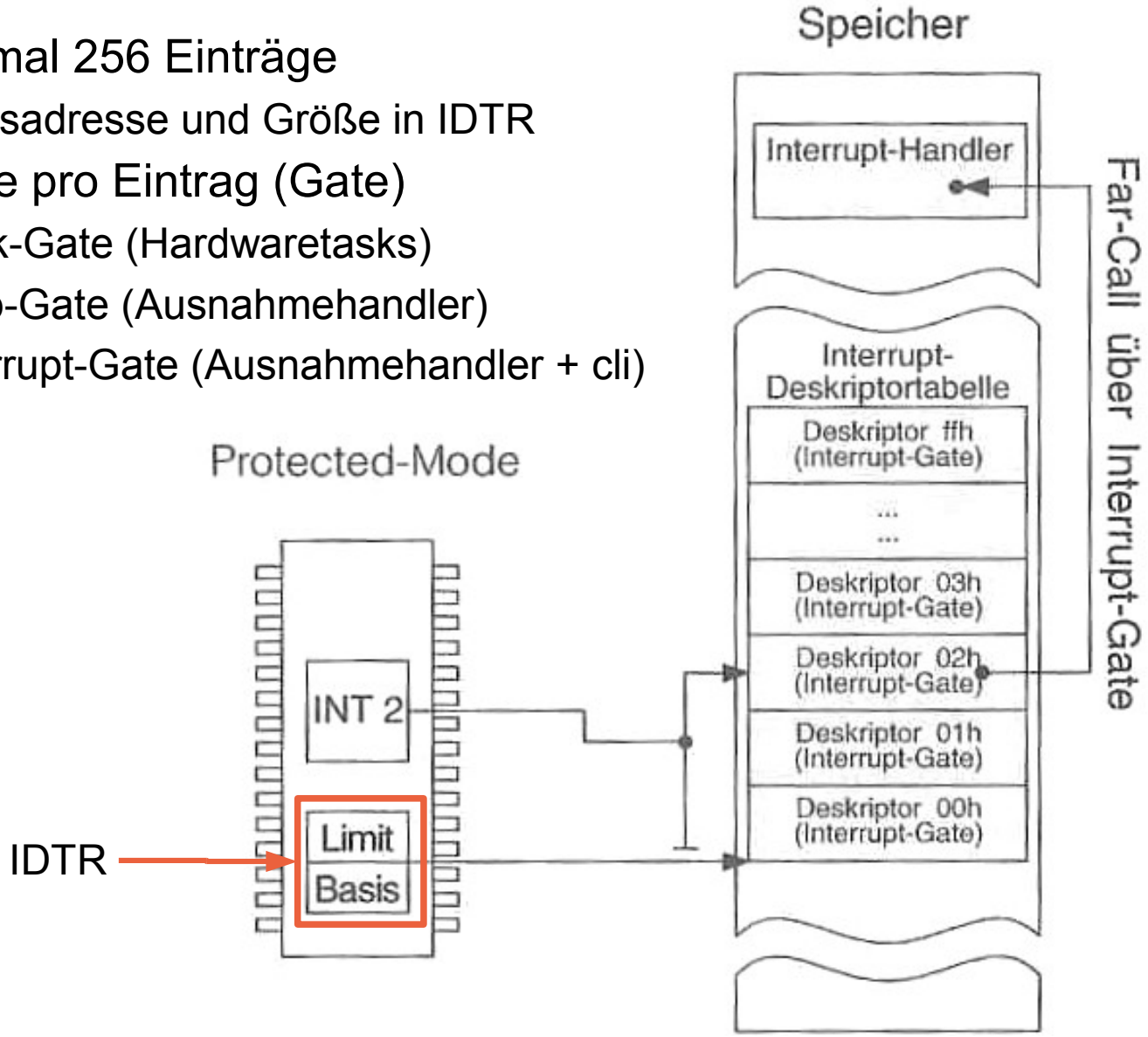
WS 07/08

Agenda: IRQ-Behandlung in OOStuBS

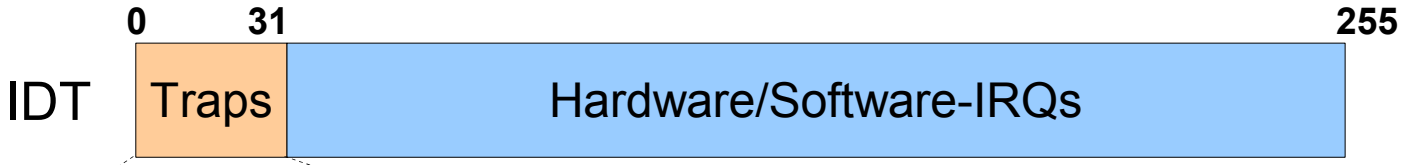
- Interrupts und Traps beim x86
  - Die Interrupt-Deskriptor-Tabelle (IDT)
  - Aufbau der IDT
  - Traps und Hardware-IRQs
- Der programmierbare Interruptcontroller PIC 8259A
  - Aufbau
  - Verwendung im PC
  - Initialisierung und Programmierung
- Interruptbehandlung in OOStuBS
  - Ablauf
  - Kontextsicherung

i386: Interrupt-Deskriptor-Tabelle (IDT)

- maximal 256 Einträge
  - Basisadresse und Größe in IDTR
- 8 Byte pro Eintrag (Gate)
  - Task-Gate (Hardwaretasks)
  - Trap-Gate (Ausnahmehandler)
  - Interrupt-Gate (Ausnahmehandler + cli)



i386-IDT: Aufbau



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

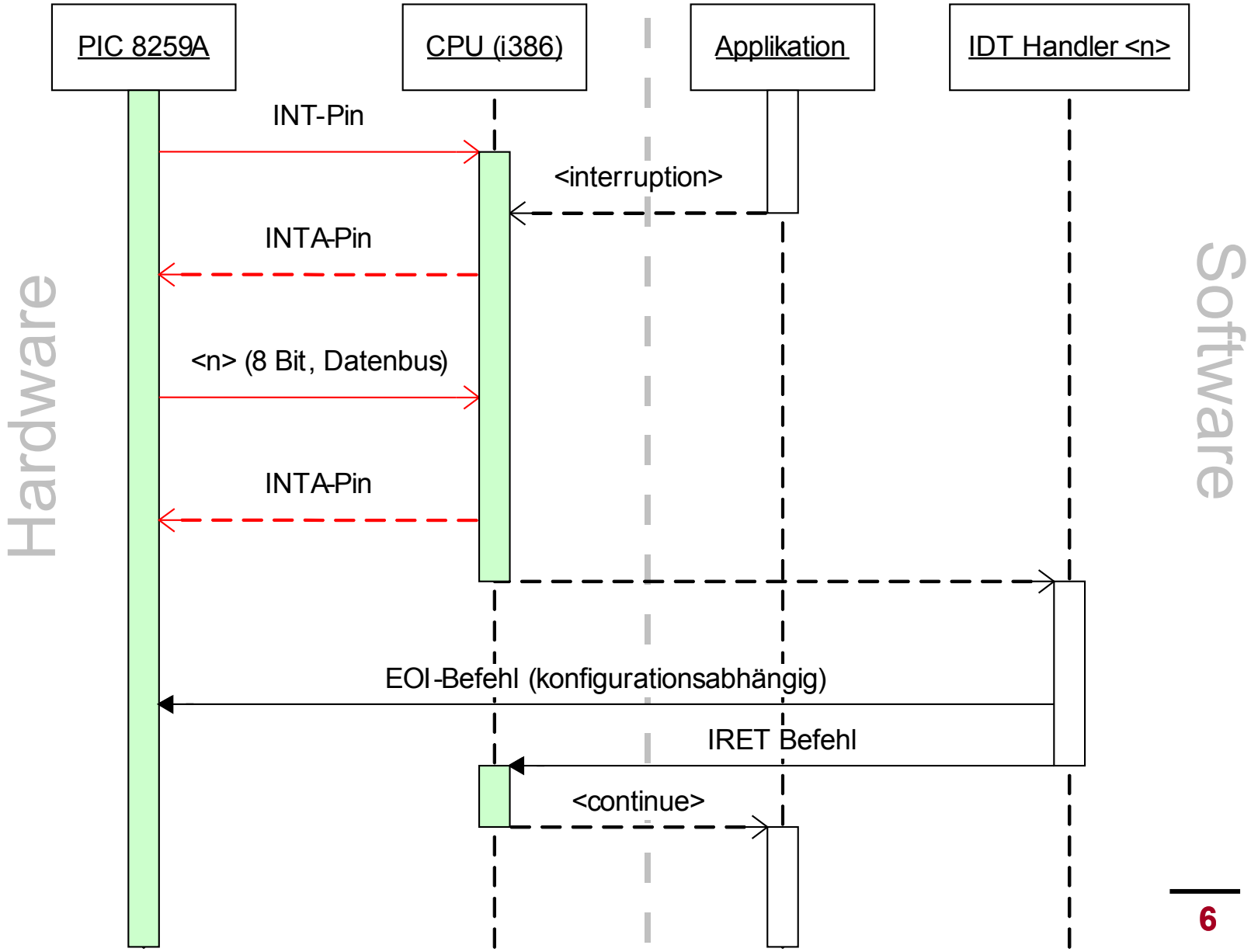
- Einträge 0-31 für Traps (fest)
- Trap = Ausnahme, die synchron zum Kontrollfluss auftritt
  - Division durch 0
  - Seitenfehler
  - Unterbrechungspunkt
  - ...
- Einträge 32-255 für IRQs (variabel)
  - Software (INT <*nummer*>)
  - Hardware (INT-Pin der CPU auf HIGH, Nummer auf Datenbus)

Hardware-IRQs bei x86-CPUs

- Bis einschließlich i486 hatten x86-CPUs nur **einen** Interrupt-Eingang (INT) plus einen NMI-Eingang
  - INT kann mit dem IE-Bit im Statuswort maskiert werden
    - cli-Befehl (clear interrupt enable) – Interruptverarbeitung sperren
    - sti-Befehl (set interrupt enable) – Interruptverarbeitung freigeben
  - NMI kann auf der CPU nicht maskiert werden (sagt ja schon der Name)
    - beim PC aber trotzdem durch externe Hardware...

- Externer Controller muss Nummer auf den Bus legen
  - Beim PC ist das der **Programmable Interrupt Controller 8259A**
  - Datenaustausch zwischen CPU und PIC 8259A erfolgt nach einem festgelegtem Protokoll

Ablauf eines Hardware-IRQ

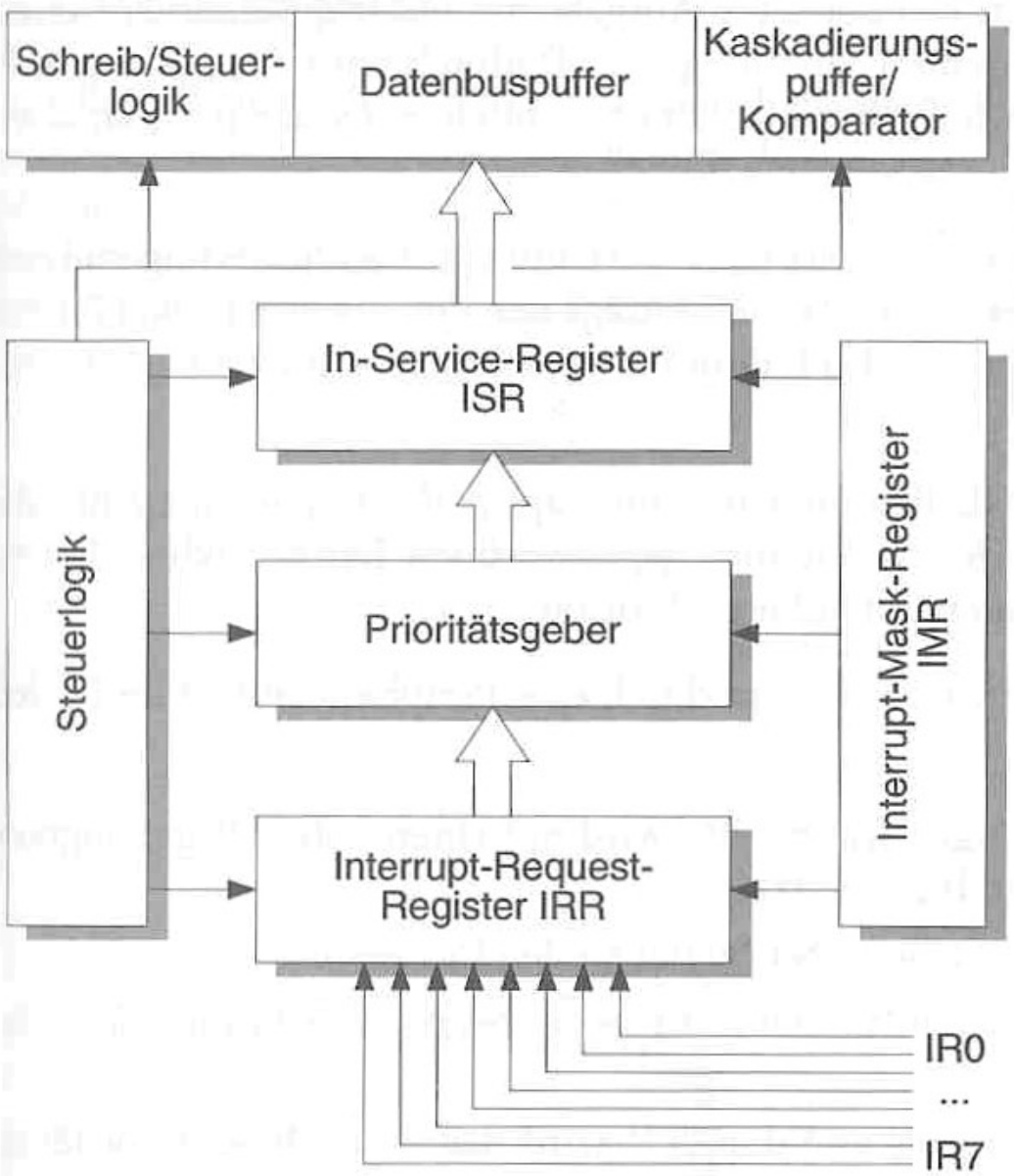


Zustandssicherung

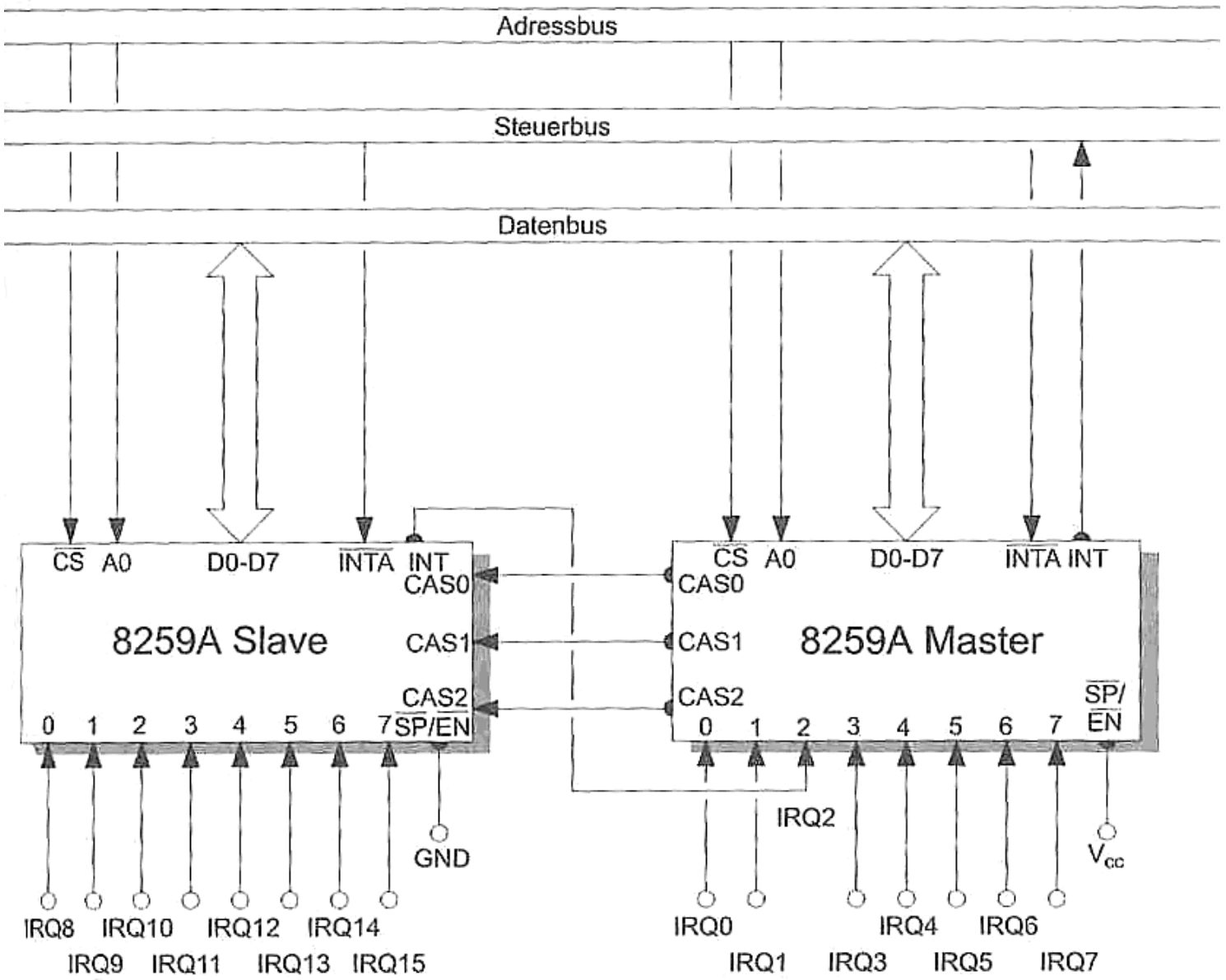
- Wenn eine Unterbrechung eintritt, sichert die CPU automatisch einen Teil des Zustands auf dem Stapel
  - condition codes (eflags)
  - aktives Codesegment (cs)
  - Rücksprungadresse (eip)

- Der automatisch gesicherte Zustand wird bei der Ausführung von iret zurückgesetzt
  - verwendet der Handler weitere Register, so muss er diese selber sichern!

PIC 8259A – Interner Aufbau

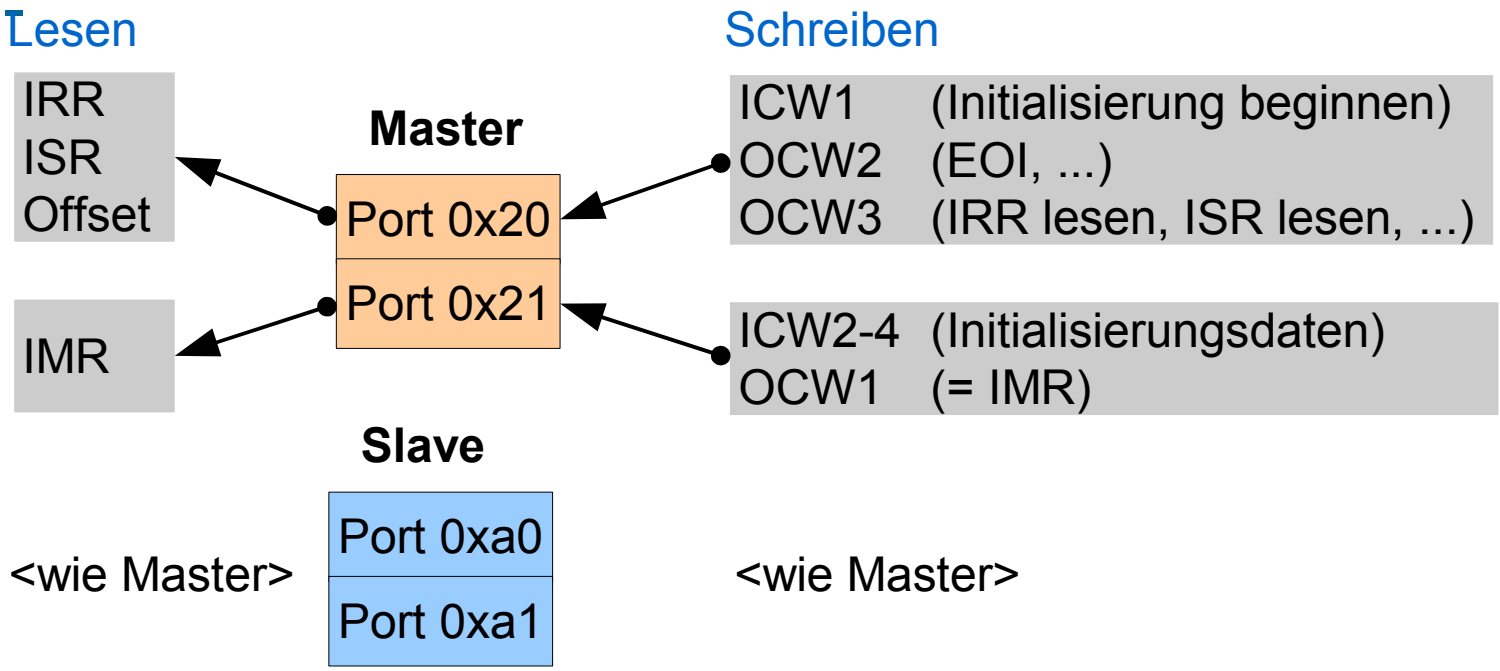


Kaskadierung im PC (15 Interrupts)



Zugriff auf die PICs über IO-Ports

- Jeder PIC hat zwei Ports, die gelesen/geschrieben werden können
- Schreibdaten: ICW1-4, OCW1-3
  - ICW = Initialization Control Word – Initialisierung des PICs
  - OCW = Operation Control Word – Kommandos im Betrieb
- Lesedaten abhängig von Kommando



Initialisierung der PICs – Teil 1

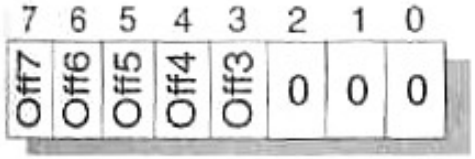
ICW1

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

OOStuBS-Einstellung:

Master	Slave
00010001	00010001

ICW2

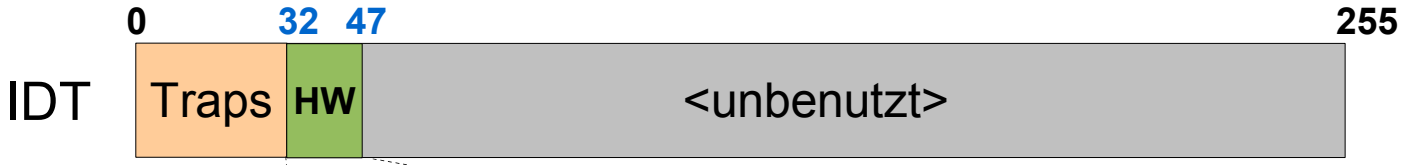


Off7..Off3: programmierbarer Offset des Interrupt-Vektors

OOStuBS-Einstellung:

Master	Slave
00100000	00101000

Mapping der HW-IRQs (OOStuBS)

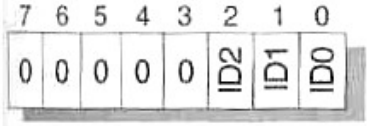


IRQ-Belegung (Standard-AT)

IRQ 0	**System Timer**
IRQ 1	Tastatur (Keyboard)
IRQ 2	PIC Kaskadierung
IRQ 3	COM 2
IRQ 4	COM 1
IRQ 5	
IRQ 6	Floppy
IRQ 7	LPT 1
IRQ 8	CMOS Echtzeituhr
IRQ 9	(HW-Mapping von IRQ 2)
IRQ10	
IRQ11	
IRQ12	PS/2
IRQ13	numerischer Coprozessor
IRQ14	1. IDE Port
IRQ15	2. IDE Port

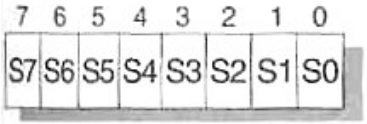
Initialisierung der PICs – Teil 2

ICW3 (Slave)



ID2..ID0: Identifizierungsnummer des Slave-PIC

ICW3 (Master)

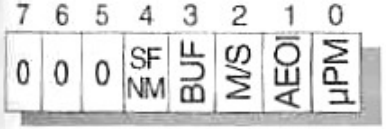


S7..S0: 0=zugehörige IR-Leitung ist mit Peripheriegerät verbunden oder frei
1=zugehörige IR-Leitung ist mit Slave-PIC verbunden

OOStuBS-Einstellung:

Master	Slave
00000100	00000010

ICW4



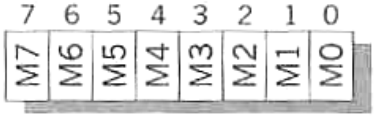
SFNM:	0=kein Special-Fully-Nested-Modus	1=Special-Fully-Nested-Modus
BUF:	0=kein gepufferter Modus	1=gepufferter Modus
M/S:	0=Slave-PIC	1=Master-PIC
AEOI:	0=manueller EOI	1=automatischer EOI
µPM:	0=Betrieb im MCS-80/85-Modus	1=Betrieb im 8086/88-Modus

OOStuBS-Einstellung:

Master	Slave
00000011	00000011

Programmierung der PICs

OCW1

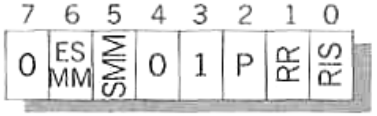


M7..M0: 0=zugehörige IRQ-Leitung ist nicht maskiert
1=zugehörige IRQ-Leitung ist maskiert

OCW2

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



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

- Interruptmaske (IMR)
  - schreiben und lesen über Port 0x21 / 0xa1

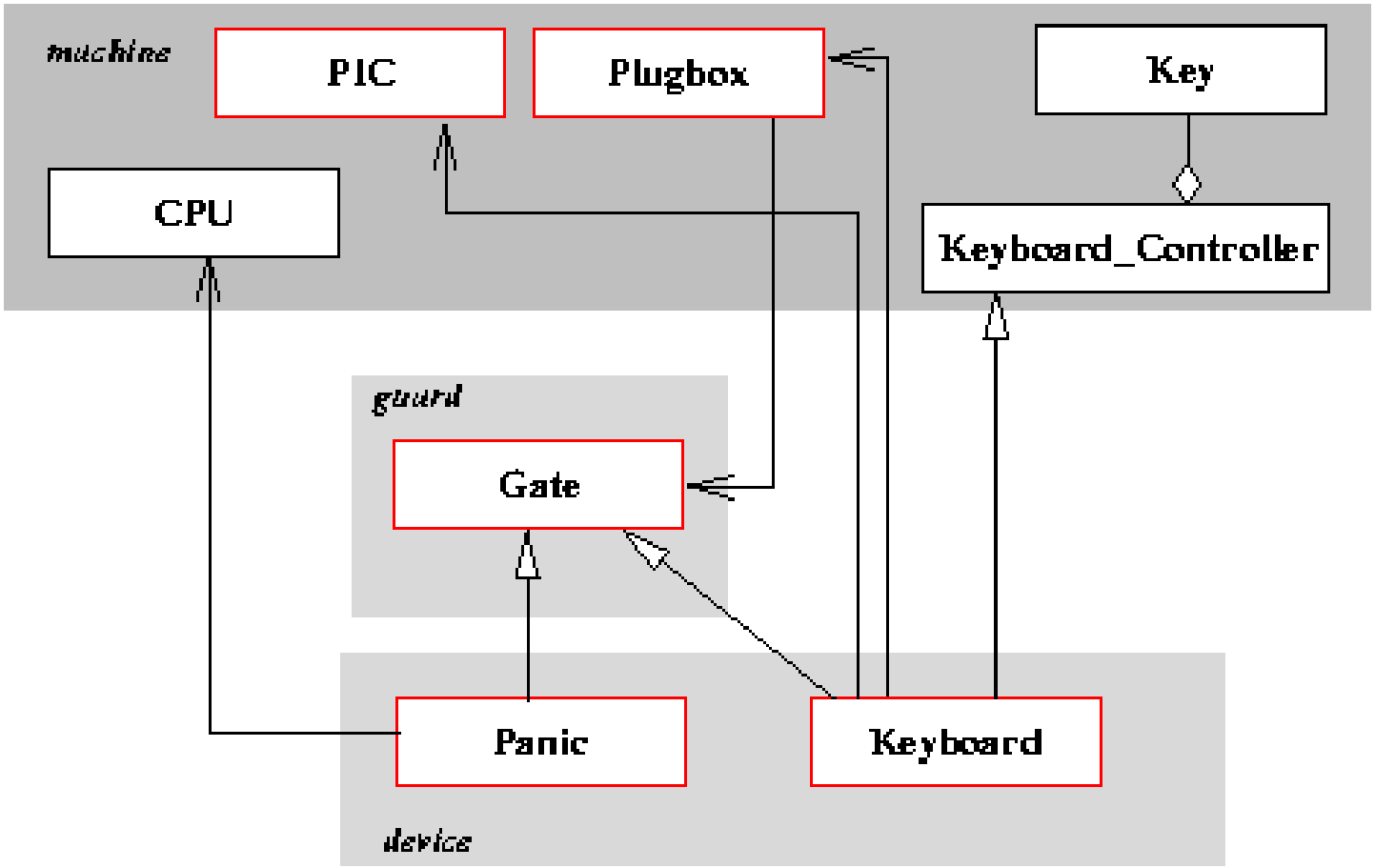
Interrupthandler in OOStuBS

- Behandlung startet in der Funktion `guardian()`
  - bekommt die IRQ-Nummer als **Parameter**:

```
void guardian(unsigned int slot) {
  ... // IRQ-Handler (Gate) aktivieren
}
```

  - Interrupts sind während der Abarbeitung gesperrt
    - können mit `sti` manuell wieder zugelassen werden
    - werden automatisch wieder zugelassen, wenn `guardian()` terminiert
- Die eigentlichen (spezifischen) IRQ-Handler
  - sind Instanzen der Klasse *Gate*
  - werden an- und abgemeldet über die Klasse *Plugbox*

Interrupthandler in OOStuBS



C-Funktionen als IRQ-Handler

- `guardian()` ist eine C-Funktion
  - erwartet die IRQ-Nummer als Parameter

- Kann `guardian()` direkt in die IDT eingetragen werden?
  - Wer sichert den Kontext?
  - Wie wird der Parameter übergeben?
  - Wo steht das `iret` zum Abschließen der IRQ-Behandlung?

⇒ Assembler-Stubs in *startup.asm*

Kontextsicherung

- Kontextsicherung:
  - eflags, cs und eip wurden bereits von der CPU gesichert
  - alle weiteren Register müssen vom IRQ-Handler gesichert werden
    - entweder im Assembler-Teil
    - oder der Compiler generiert bereits entsprechenden Code

- Kontextsicherung beim Aufruf von Funktionen
  - Lösung 1: *Aufrufende* Funktion sichert alle Register, die sie später noch braucht
  - Lösung 2: *Aufgerufene* Funktion sichert alle Register, die sie verändert
  - Lösung 3: Ein Teil der Register wird vom Aufrufer, ein anderer Teil vom Aufgerufenen gesichert

Kontextsicherung in Hochsprachen

- In der Praxis wird Lösung 3 verwendet
  - Grundsätzlich hängt das natürlich vom Compiler ab
  - CPU-Hersteller definiert jedoch Konventionen, damit Interoperabilität auf Binärcodeebene sichergestellt ist
- Register werden in 2 Subsets aufgeteilt
  - Flüchtige Register („scratch registers“)
    - Compiler geht davon aus, dass Unterprogramm den Inhalt verändert
    - Aufrufer muss Inhalt gegebenenfalls sichern
    - beim x86 sind **eax**, **ecx**, **edx** und **eflags** als flüchtig definiert
  - Nichtflüchtige Register („non-scratch registers“)
    - Compiler geht davon aus, dass der Inhalt nicht verändert wird
    - Aufgerufene Funktion muss Inhalt gegebenenfalls sichern
    - beim x86 sind alle sonstigen Register als nicht-flüchtig definiert

Interrupt-Handler müssen auch flüchtige Register sichern!