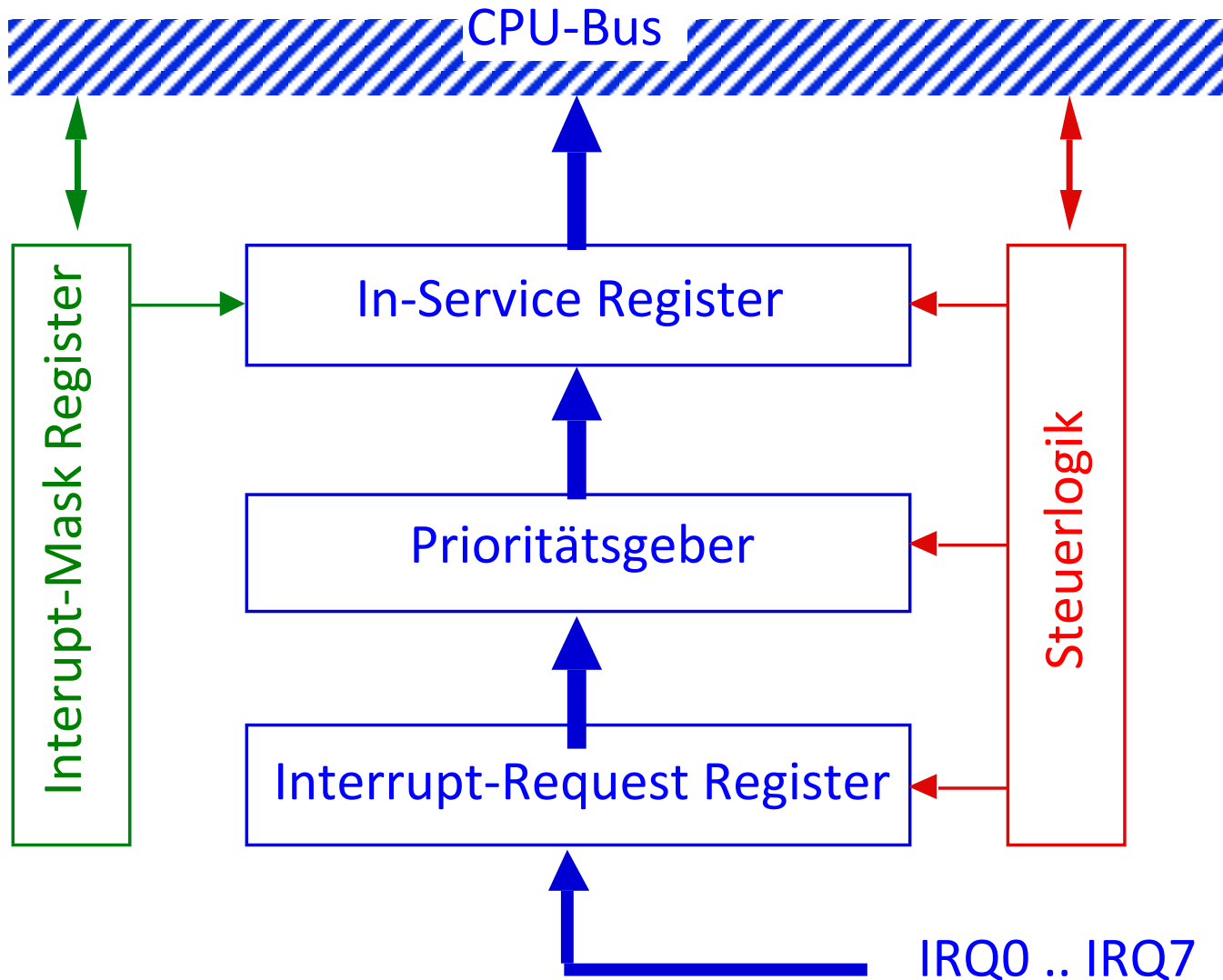


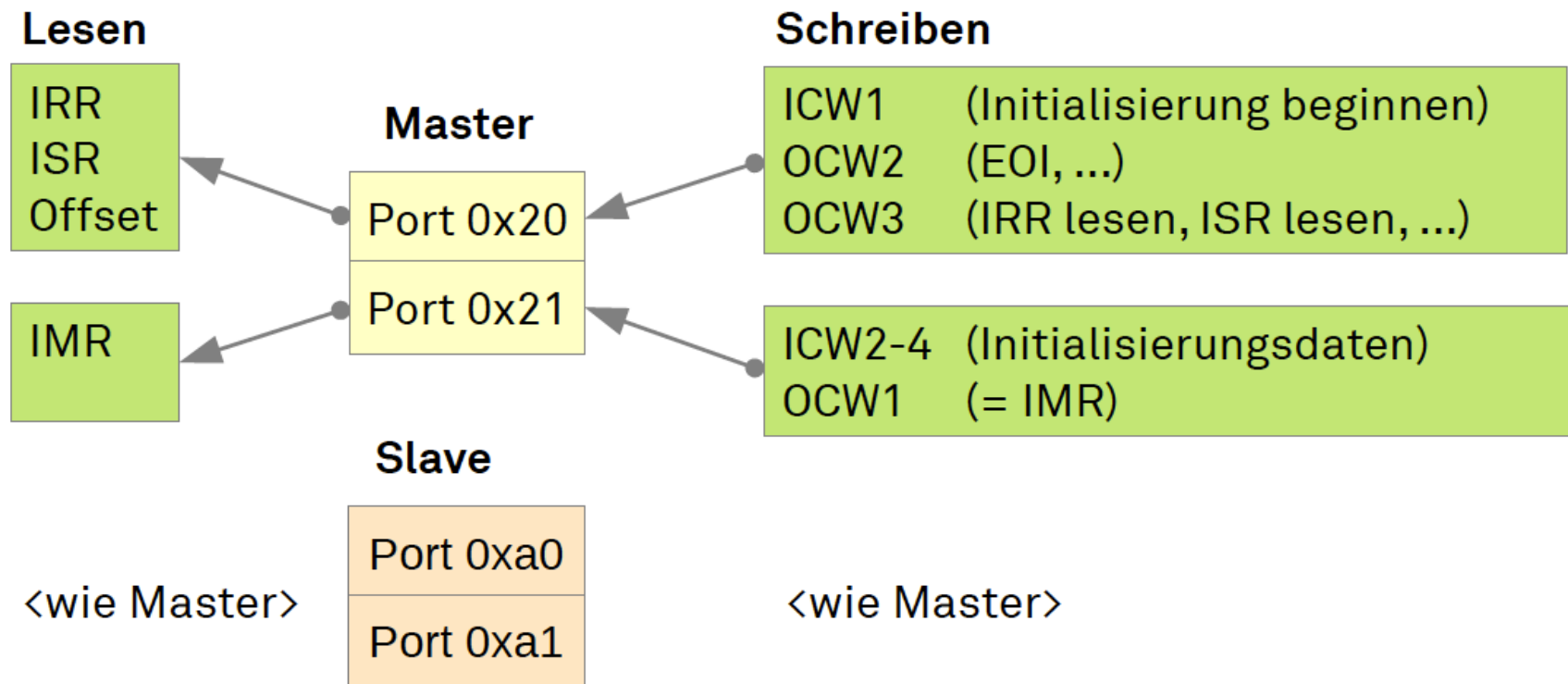
# Programmable Interrupt Controller (PIC)

## Interner Aufbau



## 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 (in startup.asm)

Initialisierung  
beginnen

ICW1

7	6	5	4	3	2	1	0
0	0	0	1	LTIM	0	SN GL	IC4

LTIM: 0=Flankentriggerung  
SNGL: 0=kaskadierte PICs  
IC4: 0=kein ICW4

1=Pegeltriggerung  
1=nur Master  
1=ICW4 notwendig

HHUos Einstellung:

Master

00010001

Slave

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

HHUos Einstellung:

Master

00100000

Slave

00101000

Offset=32

Offset=40

# Initialisierung der PICs – Teil 1 (in interrupts.asm)

Initialisierung  
beginnen

ICW1

7 6 5 4 3 2 1 0

0 0 0 1 0 0 0 1

HHUos Einstellung:

Master

Slave

00010001

00010001

; Neuprogrammierung der PICs (Programmierbare Interrupt-Controller),  
; damit alle 15 Hardware-Interrupts nacheinander in der idt liegen.

reprogram\_pics:

```
    mov     al,0x11      ; ICW1: 8086 Modus mit ICW4
    out     0x20,al
    call    delay
    out     0xa0,al
    call    delay
    mov     al,0x20      ; ICW2 Master: IRQ # Offset (32)
    out     0x21,al
    call    delay
    mov     al,0x28      ; ICW2 Slave: IRQ # Offset (40)
    out     0xa1,al
    call    delay
```

Off7..Off3: programmierbarer Offset des Interrupt-Vektors

# Initialisierung der PICs – Teil 2 (in interrupts.asm)

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)

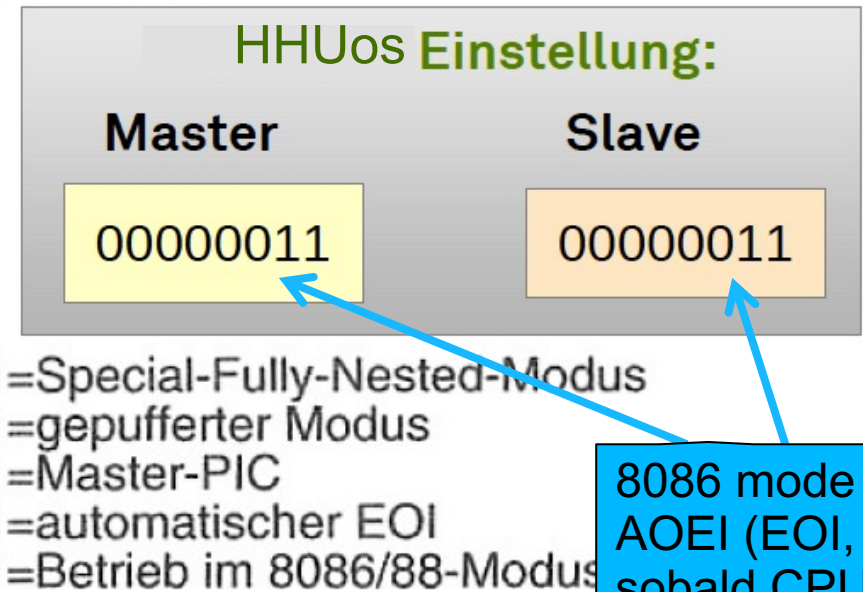
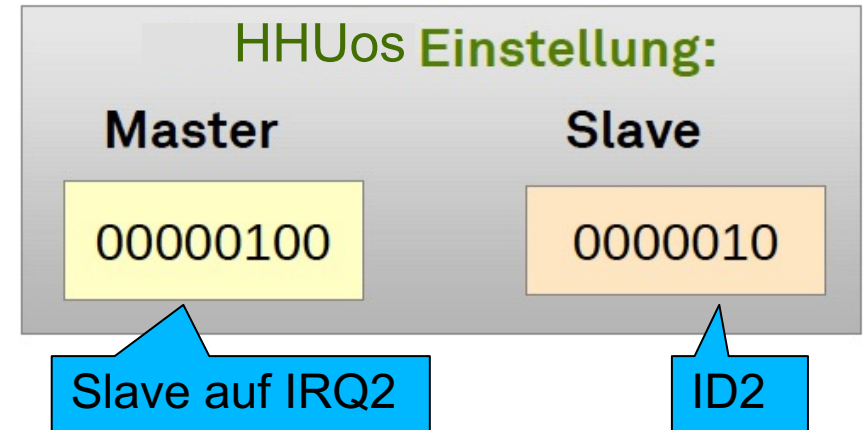
7	6	5	4	3	2	1	0
S7	S6	S5	S4	S3	S2	S1	S0

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

ICW4

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

SFNM: 0=kein Special-Fully-Nested-Modus  
BUF: 0=kein gepufferter Modus  
M/S: 0=Slave-PIC  
AEOL: 0=manueller EOI  
μPM: 0=Betrieb im MCS-80/85-Modus



1=Special-Fully-Nested-Modus  
1=gepufferter Modus  
1=Master-PIC  
1=automatischer EOI  
1=Betrieb im 8086/88-Modus

# Initialisierung der PICs – Teil 2 (in interrupts.asm)

ICW3 (Slave)

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

HHUos **Einstellung:**

Master

Slave

```

...
mov     al,0x04      ; ICW3 Master: Slaves an IRQs
out     0x21,al
call    delay
mov     al,0x02      ; ICW3 Slave: Verbunden mit IRQ2 des Masters
out     0xa1,al
call    delay
mov     al,0x03      ; ICW4: 8086 Modus und automatischer AEOI
out     0x21,al
call    delay
out     0xa1,al
call    delay
...

```

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

1=Special-Fully-Nested-Modus  
 1=gepufferter Modus  
 1=Master-PIC  
 1=automatischer EOI  
 1=Betrieb im 8086/88-Modus

8086 mode mit  
 AEOI (EOI,  
 sobald CPU  
 IRQ bedient



# Programmierung der PICs

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

## Interruptmaske (IMR)

- schreiben und lesen über Port 0x21 / 0xa1

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 setzen  
111: bei spezifischem EOI-Befehl rotieren

Wichtig für  
Übungsblatt 4

OCW3

7	6	5	4	3	2	1	0
0	ES MM	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:         | 0=kein Polling         | 1=Polling-Modus       |

# Interruptbehandlung in hhuTOS

## IDT-Aufbau (in interrupts.asm)

```
[SECTION .data]
```

```
; Interrupt Descriptor Table mit 256 Einträgen
```

```
idt:
```

```
%macro idt_entry 1
```

```
dw (wrapper_%1 - wrapper_0) & 0xffff ; Offset 0 .. 15
```

```
dw 0x0000 | 0x8 * 2 ; Selector zeigt auf den 64-Bit-Codesegment-Deskriptor der GDT
```

```
dw 0x8e00 ; 8 -> interrupt is present, e -> 80386 32-bit interrupt gate
```

```
dw ((wrapper_%1 - wrapper_0) & 0xffff0000) >> 16 ; Offset 16 .. 31
```

```
dd ((wrapper_%1 - wrapper_0) & 0xffffffff00000000) >> 32 ; Offset 32..63
```

```
dd 0x00000000 ; Reserviert
```

```
%endmacro
```

```
%assign i 0
```

```
%rep 256
```

```
idt_entry i
```

```
%assign i i+1
```

```
%endrep
```

```
idt_descr:
```

```
dw 256*8 - 1 ; 256 Einträge
```

```
dq idt
```

Template für einen IDT-Eintrag, je 8 Byte (Trap Gate)

256 IDT-Einträge erzeugen

Limit (16 Bit)  
Base address (64-Bit)



# Interruptbehandlung in hhuTOS

## IDT-Aufbau (in interrupts.asm)

```
[SECTION .data]
```

```
; Interrupt Descriptor Table mit 256 Einträgen
```

```
idt:
```

```
%macro idt_entry 1
    dw (wrapper_%1 - wrapper_0)
    dw 0x0000 | 0x8 * 2
    dw 0x8e00 ; 8 -> int
    dw ((wrapper_%1 - wrapper_0) << 16)
    dd ((wrapper_%1 - wrapper_0) << 32)
    dd 0x00000000 ; Reserved
%endmacro
```

```
%assign i 0
```

```
%rep 256
```

```
    idt_entry i
```

```
%assign i i+1
```

```
%endrep
```

```
idt_descr:
```

```
    dw 256*8 - 1 ; 256*8 - 1
```

```
    dq idt
```

```
setup_idt:
```

```
    mov     rax, wrapper_0
```

```
; Bits 0..15 -> ax, 16..31 -> bx, 32..64 -> edx
```

```
    mov     rbx, rax
```

```
    mov     rdx, rax
```

```
    shr     rdx, 32
```

```
    shr     rbx, 16
```

```
    mov     r10, idt ; Zeiger auf das aktuelle Interrupt-Gate
```

```
    mov     rcx, 255 ; Zähler
```

```
.loop:
```

```
    add     [r10+0], ax
```

```
    adc     [r10+6], bx
```

```
    adc     [r10+8], edx
```

```
    add     r10, 16
```

```
    dec     rcx
```

```
    jge     .loop
```

```
    lidt    [idt_descr]
```

```
    ret
```

Basisadresse v. wrapper\_0  
auf alle IDT-Einträge  
aufaddieren

IDTR laden

# Interruptbehandlung in HHUos

## 256 Wrapper-Routinen: Vektor-Nummer ermitteln (in startup.asm)

**; Spezifischer Kopf der Unterbrechungsbehandlungsroutinen**

**%macro wrapper 1**

**wrapper\_%1:**

**push** rbp

**mov** rbp, rsp

**push** rax

**mov** al, %1

**jmp** wrapper\_body

**%endmacro**

Pro IDT-Eintrag eine wrapper-Routine. Diese speichert die Vektor-Nummer in `rax`. Damit kann später die Vektor-Nummer ermittelt werden.

**; ... wird automatisch erzeugt.**

**%assign i 0**

**%rep 256**

**wrapper i**

**%assign i i+1**

**%endrep**

# Interruptbehandlung in HHUos

## Zustandssicherung & Interrupt-Delegation an C-Code

```
[EXTERN int_disp] ; Funktion in C, welche Interrupts behandelt
```

```
; Gemeinsamer Rumpf
wrapper_body:
; Das erwartet der gcc so
cld
; Flüchtige Register sichern
push rcx
push rdx
push rdi
push rsi
push r8
push r9
push r10
push r11
```

Import der Hochsprachen-  
Funktion im Assembler-Code

```
; Der generierte Wrapper liefert nur 8 Bit
and rax, 0xff
```

```
; Nummer der Unterbrechung als Argument übergeben
```

```
mov rdi, rax
call int_disp
```

Funktion in der  
Hochsprache aufrufen

```
; Flüchtige Register wiederherstellen
```

```
pop r11
pop r10
pop r9
pop r8
pop rsi
pop rdi
pop rdx
pop rcx
```

```
; ... auch die aus dem Wrapper
```

```
pop rax
pop rbp
```

```
; Fertig!
iretq
```

## Weitere Informationen

- Im Datenblatt: 8259A.pdf (in git)
- Und hier: [http://wiki.osdev.org/8259\\_PIC](http://wiki.osdev.org/8259_PIC)