Interrupts in hhuTOS

IDT & Handler

- Für die IDT werden im Assemblerteil 256 Einträge per Macros erzeugt
- Für jede Vektor-Nummer wird eine eigene kleine Assembler-Funktion `wrapper_x` (x = {0, …, 255}) per Macro generiert.
- Alle `wrapper_x` Funktionen speichern die Vektornummer in `rax` und rufen dann `wrapper_body`
- In `wrapper_body` werden die Register gesichert und `int_disp` in der Hochsprache aufgerufen, dann die Register wiederhergestellt und per `iretq` zum `wrapper_x` zurückgesprungen
- Der Rücksprung von `wrapper_x` beendet dann die Interrupt-Behandlung

Erzeugung der 256 IDT-Einträge

Auszug aus `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 64-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 (16 Byte), hier „nur“ Interrupt Gates

256 IDT-Einträge erzeugen

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

Erzeugung der 256 Interrupt-Handler

```
; Spezifischer Kopf der Unterbrechungsbehandlungsroutinen
%macro wrapper 1
wrapper_%1:
  push   rbp
  mov    rbp, rsp
  push   rax
  mov    al, %1
  jmp    wrapper_body
%endmacro

; ... wird automatisch erzeugt.
%assign i 0
%rep 256
wrapper i
%assign i i+1
%endrep
```

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

Aufruf des Interrupt-Handlers in der Hochsprache

```
[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

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

  ; Nummer der Unterbrechung als Argument übergeben
  mov    rdi, rax
  call   int_disp

  ; 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
```

Import der Hochsprachen-Funktion im Assembler-Code

Funktion in der Hochsprache aufrufen

Initialisierung zur Laufzeit

- In der Einstiegsfunktion der Hochsprache muss möglichst früh `init_interrupts` aufgerufen werden
- Hier wird zuerst die IDT und dann der PIC initialisiert

```
_init_interrupts:
    call setup_idt
    call reprogram_pics
    ret
```

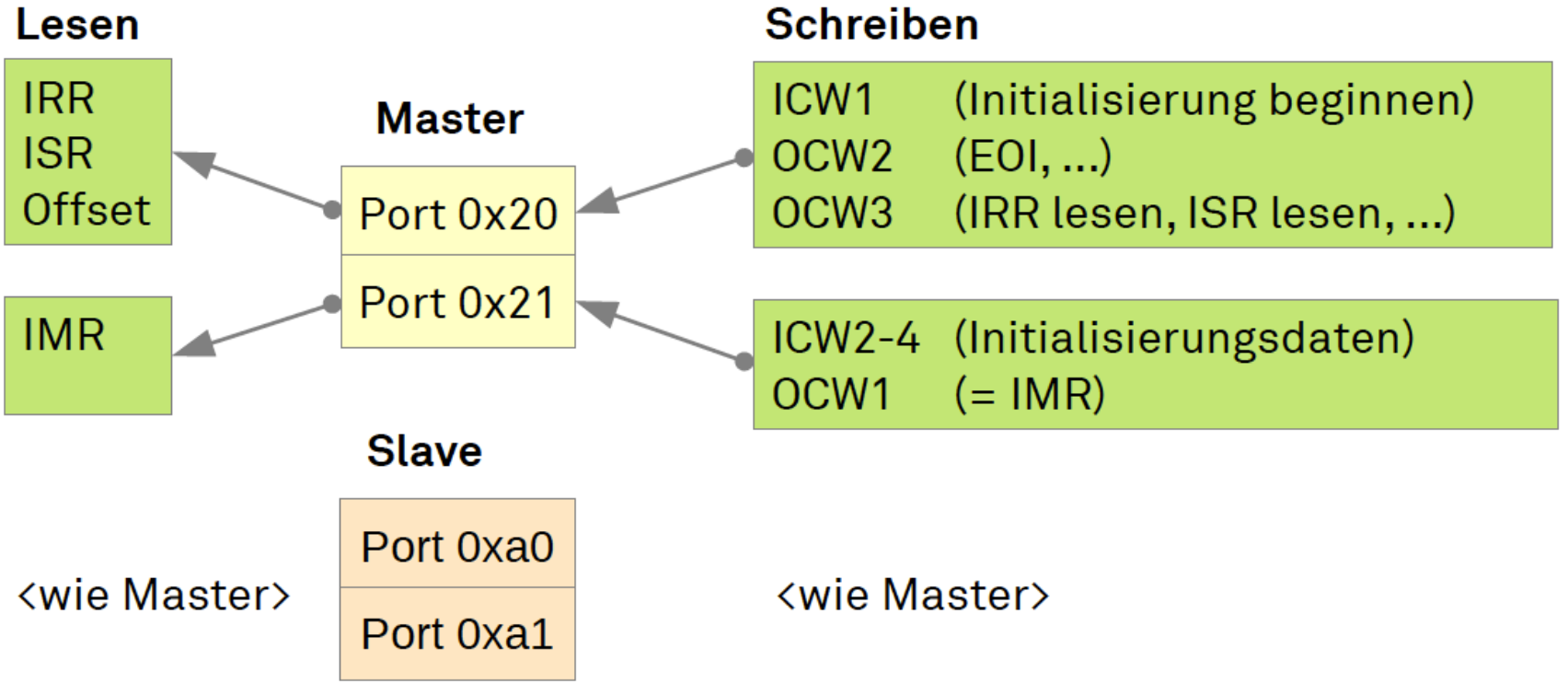
```
setup_idt:
  mov    rax, wrapper_0

  ; 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 von `wrapper_0` auf alle IDT-Einträge aufaddieren

IDTR laden

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 `interrupts.asm`)

Initalisierung beginnen

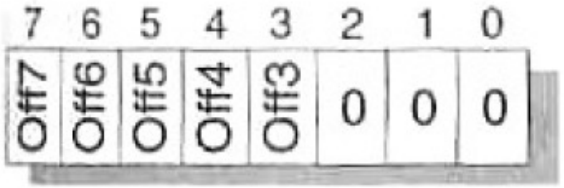
ICW1

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

hhuTOS: Einstellung

Master	Slave
00010001	00010001

ICW2



Off7..Off3: programmierbarer Offset des Interrupt-Vektors

hhuTOS: Einstellung

Master	Slave
00100000	00101000
Offset=32	Offset=40

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

hhuTOS: Einstellung

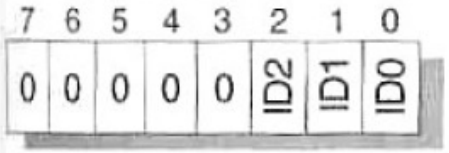
Master	Slave
00010001	00010001

```
; 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

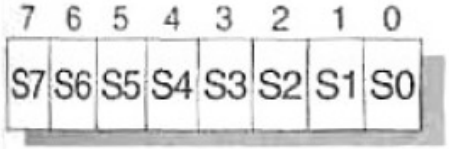
Initialisierung der PICs – Teil 2 (in `interrupts.asm`)

ICW3 (Slave)

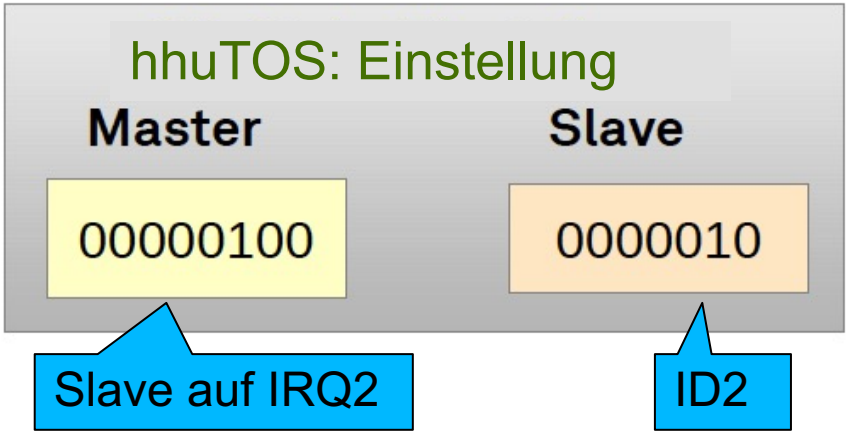


ID2..ID0: Identifizierungsnummer des Slave-PIC

ICW3 (Master)

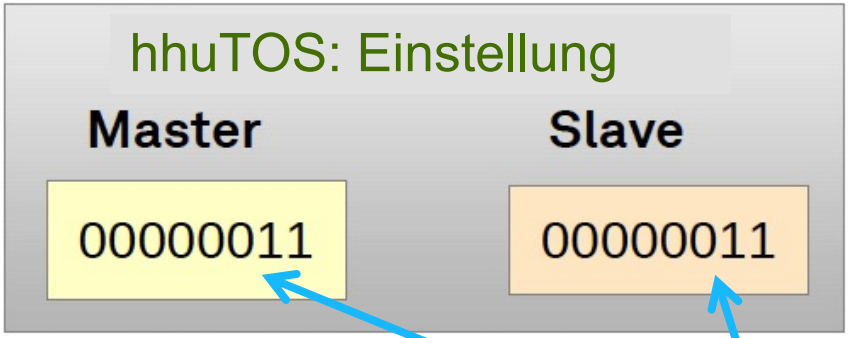


S7..S0: 0=zugehörige IR-Leitung ist mit Peripheriegerät verbunden oder frei
1=zugehörige IR-Leitung ist mit Slave-PIC verbunden



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



8086 mode mit AOEI (EOI, sobald CPU IRQ bedient

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	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

8086 mode mit AOEI (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

Wichtig für Aufgabe 4

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

Zusammenfassung

- Es gibt 256 verschiedene Interrupt-Handler `wrapper_0` bis `wrapper_255` (damit wir die Vektornummer ermitteln können)
- Diese Handler rufen alle `wrapper_body` auf
  - Hier werden die Register gesichert, die Hochsprachenfunktion `int_disp` gerufen und die Register danach wiederhergestellt
- Der PIC wird initialisiert, sodass der End-Of-Interrupt automatisch erzeugt wird und alle IRQs sind maskiert.

Weitere Informationen

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