

Interrupts

In vielen Fällen möchte man Unterprogramme aufrufen, wenn ein bestimmtes Ereignis eingetreten ist.

```
while (1) {
    if (event()) {
        handle_event();
    } else {
        do_normal_work();
    }
}
```

Polling

- kostet Rechenzeit
- hat lange Reaktionszeit
- umständlich zu programmieren (besonders in Multi-User-Systemen!)

=> Interrupts

Interrupts

Aufruf eines „handle_event“-Unterprogramms automatisch durch CPU bei Signalisierung eines „event“-Interrupts.

Über Tabelle einstellbar

- welche Ereignisse (Interrupts)
- welche Unterprogramme (Interrupt-Handler)

anstoßen sollen (Beispiele):

Taste gedrückt => Tastatur-Lese-Unterprogramm
Netzwerk-Paket angekommen => Netzwerk-Controller-Ausleseprozedur
Hardware-Uhr getickt => Uhrzeit-Ausleseprozedur
...

die Interrupts werden durch Zahlen codiert (Tabellenindex)

Adressen der Interrupt-Handler (Interrupt-Vektoren) werden in Tabelle gespeichert

Tabelle liegt an festgelegter Stelle im Hauptspeicher (z.B. Adresse 0)

Interrupt-Handler

Interrupt-Handler sind

- „normale“ Unterprogramme
- ohne Parameter
- ohne Rückgabewert.

Sie dürfen Register nicht verändern (sonst stören sie den Ablauf des normalen Programms).

Daher speichern Interrupt-Handler meist Register zu Beginn auf dem Stack und restaurieren sie vor der Rückkehr zum normalen Programmablauf.

CPUs sichern/restaurieren einige/alle Register automatisch

=> iret-Befehl (statt ret; i80x86) bzw. rti (statt rts; m68x05)

i80x86 sichert Program-Counter, Condition-Code-Register

m68x05 sichert Program-Counter, Condition-Code-Register, Index-Register und Akku

Interrupt-Handler

Beispiel (i80x86):

```
                                /* Int.-Handler-Tabelle */
0x0000          0x08343248
0x0004          0x0834af12    /* Adressen der Int.-Handler */
0x0008          0x0834bc00
                ...
0x083eaf12:    pushl %eax    /* Int.-Handler */
                ...
                ...
                popl %eax
                iret
```

i80x86-CPU sichert/restauriert %eip und %eflags automatisch (=> Condition-Codes dürfen verändert werden)

Interrupt-Handler sichert und restauriert %eax (=> %eax darf verwendet werden)

Interrupts (Interrupt-Masken)

In einigen Programm-Bereichen ist es nicht erwünscht, dass das normale Hauptprogramm durch Interrupts unterbrochen wird.

- zeitkritische Bereiche
(Unterbrechungen aus Zeitgründen grundsätzlich unerwünscht)

Beispiel:

Einlesen von Messwerten zu bestimmten Zeiten

- Nebenläufigkeits-kritische Bereiche
(während das Hauptprogramm einen kritischen Abschnitt durchläuft, darf ein Interrupt-Handler nicht einen bestimmten anderen kritischen Abschnitt durchlaufen)

Beispiel:

Hauptprogramm liest Tastatur-Puffer aus

Interrupt-Handler will neue Zeichen in den Puffer eintragen

=> Möglichkeiten notwendig, Interrupts zu sperren und wieder freizugeben (zu maskieren)

Interrupts (Interrupt-Masken)

Interrupts sperren und freigeben geschieht z.B. über

- Interrupt-Controller
(jeder Interrupt kann einzeln freigegeben oder gesperrt werden)
- Interrupt-Enable/Disable-Bit in CPU
(alle Interrupts werden gleichzeitig freigegeben bzw. gesperrt)
 - i80x86: sti, cli
 - m68x05: cli, sei
- Interrupt-Priorität in CPU
(Interrupts unter einer Priorität werden gesperrt; alle höheren sind erlaubt)
 - m680xx

Kombinationen sind möglich (z.B. IBM-PC: Interrupt-Controller + Interrupt-Enable-Bit)

Interrupts

Ein Unterprogramm, das ein Zeichen aus einem Tastatur-Puffer entnimmt besitzt daher z.B. eine Struktur wie folgt:

```
getkey: cli                                /* disable interrupts */
        movb 433, %bl                      /* read character from buffer */
        movb 400(%bl), %al
        addb $1, %bl
        andb $0x1f, %bl
        movb %bl, 433

        sti                                /* enable interrupts */
        ret
```

Hinweis:

Da ein cli-Befehl u.U. den gesamten Rechner lahmlegen kann, ist er nur im Supervisor-Mode der CPU erlaubt => nicht in „normalen“ Linux-Programmen verwendbar.

Exceptions

Während ein Programm läuft, können u.U. Fehler auftreten. Zum Beispiel:

- Programm versucht, durch Null zu teilen
- Programm versucht, auf nicht vorhandene Speicherzellen zuzugreifen
- Addition verursacht Überlauf
- ...

Fehler können durch Überprüfung der Operanden vor Ausführung eines Befehls (zum Teil abgefangen werden (z.B. vor Division erst Divisor auf Null abfragen). Fehleranfällig, mühsam, zeitaufwändig, ...

Exceptions

CPU kann die Operanden während der normalen Programmausführung testen und gegebenenfalls eine Ausnahmebehandlung durchführen.

Über Tabelle einstellbar

- welche Ausnahmen (Exceptions)
- welche Unterprogramme (Exception-Handler)

anstoßen sollen (Beispiel):

Division durch Null => Fehlermeldung an Benutzer
Additions-Überlauf => Register auf größt-möglichen Wert setzen
Speicher antwortet nicht => Speichertest durchführen
...

die Exceptions werden durch Zahlen codiert (Tabellenindex)

Adressen der Exception-Handler (Exception-Vektoren) werden in Tabelle gespeichert

Tabelle liegt im Hauptspeicher (meist gemeinsame Tabelle mit Interrupts)

Exception-Handler

Exception-Handler sind

- „normale“ Unterprogramme
- z.T. mit Parametern
- ohne Rückgabewert (aber meist mit Nebeneffekten).

Sie dürfen Register u.U. verändern (der normale Ablauf des Programms ist sowieso gestört), um den weiteren Ablauf des Programms zu ermöglichen.

Exception-Handler (Beispiel)

Hauptprogramm:

```
...
movl %esp, errorsp /* save state of stack */
call test          /* call test procedure */
error:            ...
...

```

Test-Unterprogramm:

```
test:            ... /* test something */
movl $0, %eax    /* no error so far */
ret

```

Exception-Handler:

```
exception:      movl $1, %eax    /* got error */
                movl errorsp, %esp /* restore stack */
                jmp error    /* break test */

```

System-Calls

Prinzipiell könnte man Systemaufrufe durch `call system(o.ä.)` realisieren. Notwendig wäre dazu eine bekannte (und feste!) Adresse der System-Funktionen.

Anderer Ansatz: man provoziert eine „Ausnahme“. Der Exception-Handler (Teil des Betriebssystems) ruft dann die eigentliche Systemfunktion auf. Dafür existieren bei den meisten (größeren) CPUs Extra-Befehle:

- i80x86: `int $...`
- m680x0: `trap #...`
- m68x05: `swi`