Der Umgang mit Interrupts

Ein Rechner, der Signale von außerhalb entgegenzunehmen und darauf zu reagieren hat, kann prinzipiell zwei Strategien verfolgen:

- Polling-Betrieb
  Der Rechner behält die Eingabeleitung ständig im Auge. Dadurch kann der Rechner nebenher keine weiteren Aufgaben verfolgen. Die Programmierung ist einfach und die Reaktionszeit äußerst kurz.
- Interrupt-Betrieb
  Der Rechner kümmert sich überhaupt nicht um das Eingangssignal und erledigt sein normales Programm. Das einlaufende Signal löst einen Interrupt aus. Dann unterbricht der Rechner sein Programm, kümmert sich um das Eingangssignal und setzt danach das unterbrochene Programm fort. Die Programmierung ist etwas schwieriger, aber der Rechner wird nicht blockiert.

Im Folgende möchte ich vorstellen, wie ein eigener Interrupt-Handler auf Signale von außen reagiert und wie die Software dafür aussieht.

```
program Int;
uses  dos;
const IRQNr = 5;
var   ISRold : procedure;

{$F+}
procedure ISRnew; interrupt;
begin
  { ISRold }
  { hierhin eigene Programmierung }
  port[$20] := 32 ;
end;
{$F-}

begin
  GetIntVec(IRQNr+8,addr(ISRold));
  SetIntVec(IRQNr+8,addr(ISRnew));
  port[$21] := port[$21] and not (1 shl IRQNr);
  { hierhin das normale Programm }
  SetIntVec(IRQNr+8,addr(ISRold));
  port[$21] := port[$21] or (1 shl IRQNr);
end.
```

Dieses Minimalprogramm setzt einen eigenen Interrupt-Handler in Gang:

```
program Int;
```

Diese Zeile nennt lediglich das Programm beim Namen.

```
uses  dos;
```

Die Unit ‘dos’ stellt einige Funktionen und Prozeduren zur Verfügung, die bei der Interrupt-Programmierung benutzt werden. Folglich muß diese Prozedur eingebunden werden.

```
const IRQNr = 5;
```

Ein PC verfügt über eine bestimmte Anzahl von Interrupts, die mit IRQ# bezeichnet werden. Anfangs hatten die PCs in XT-Ausführung nur 8 Interruptleitungen zur Verfügung, die zu dem Interrupt-Controller 8259 führen. Bei den AT-Rechnern sind zwei dieser Controller kaskadiert, so daß 15 Interrupts zur Verfügung stehen.
Die Verwendung des IRQ5 in diesem Programm ist beispielhaft. Ein Blick in die BIOS-Einstellung zeigt, welche Interrupts frei sind. Unter Windows wird das schwierig, hier werden etliche Interrupts mehrfach verwendet.
Der IRQ5 liegt auf dem ISA-Bus, so daß man an dieser Stelle einen Draht festmachen kann, um das Signal auf den Rechner zu führen.

```
var   ISRold : procedure;
```
Der Variablen `ISRold` wird der Typ `procedure` zugeordnet. Die Variable dient der Bezeichnung des bisherigen Interrupt-Handlers, der der gewählten Interrupt-Nummer zugeordnet ist.

```
{$F+}
```
Dem Compiler wird mitgeteilt, daß die Zeiger, die auf diese Prozedur zeigen, vom Typ ‚far' sind. Der Adressbereich ist nicht mehr beschränkt, so daß die Prozedur überall erreicht wird.

```
procedure ISRnew; interrupt;
```
Diese Prozedur beinhaltet die Programmierung des eigenen Interrupt-Handlers, oder auch Interrupt-Service-Routine ISR. Sie ist vom Typ ‚Interrupt' und kann nicht vom Programm aufgerufen werden.

```
begin
  { ISRold }
```
Der Aufruf `ISRold` ist auskommentiert, da er nur dann aufgerufen werden muß, wenn der Interrupt IRQ5 schon eine Interrupt-Service-Routine ISR enthält. Dann würde an dieser Stelle erst die bisherige ISR bearbeitet und dann mit der eigenen, verlängerten Routine fortgesetzt.

```
  { hierhin eigene Programmierung }
```
Hierher kommt der eigene Programmtext, was die ISR überhaupt tun soll.

```
  port[$20] := 32;
```
Mit diesem Portzugriff wird ein unspezifischer EOI (End of Interrupt) gesendet. Damit wird der Interrupt-Controller freigegeben, so daß er den nächsten Interrupt entgegennehmen kann.

```
end;
```
Ende der eigenen ISR.

```
{$F-}
```
Die Compileroption ‘far' wird wieder abgestellt.

Und jetzt zum Hauptprogramm. Im ersten Teil wird der Interrupt vorbereitet und zugelassen:

```
  GetIntVec(IRQNr+8,addr(ISRold));
```
Die Einsprungadresse der ISRold, der bisherigen ISR, wird gerettet, weil sie nach dem Programm wieder hergestellt werden muß. Diese Adresse wird aus der Interrupt-Vektor-Tabelle herausgelesen.
Die Adresse des Interrupts ist die Interrupt-Nummer + 8. In Tabellen hierfür findet man für den IRQ5 daher die Adresse $0d in hexadezimaler Schreibweise.

addr ist der Pascal-Befehl, um die Einsprungadresse von Prozeduren zu lesen. GetIntVec ist der Zugriff auf die Interrupt-Vektor-Tabelle.

```
SetIntVec(IRQNr+8,addr(ISRnew));
```

Dort, wo früher die Einsprungadresse der bisherigen ISR stand, wird jetzt die Adresse der Prozedur ISRnew, unserer eigenen ISR, eingetragen. Ab jetzt steht diese ISR zum Aufruf bereit.

```
port[$21] := port[$21] and not (1 shl IRQNr);
```

Das ist ein Schreibzugriff auf das Operational-Control-Word 1 (OCW1) des Interrupt-Controllers 8259. Die Bits maskieren die Interrupts und schalten sie ab. Damit ein Interrupt überhaupt vom Controller an den Prozessor durchgelassen wird, muß er ‚demaskiert' werden, d. h. das entsprechende Bit muß auf null gesetzt werden. Dieser Zugriff ist nur erforderlich, wenn der entsprechende Interrupt gesperrt war.
Bit 0 demaskiert den IRQ0, Bit 1 den IRQ1, usw. Deshalb kann mit dem Befehl ‚shl' (shift left) ein Bit ‚1' um soviele Stellen nach links geshiftet werden, wie es der IRQNr entspricht. In diesem Fall ist IRQNr = 5, damit wird aus 1 shl 5 eine 32. Mit dem Befehl ‚and not' wird das entsprechende Bit unabhängig von den anderen Bits gelöscht.
Liest man dieses OCW1 vor dem Programm aus und gibt den Wert aus, so findet man darin ebenfalls unbenutzte Interrupt-Nummern. (In meinem Fall habe ich eine 184 ausgelesen, das ist das Bitmuster 10111000. Die Einsen sind unbenutzte, maskierte Interrupts, somit sind die Interrupts 3, 4, 5 und 7 frei.)

```
SetIntVec(IRQNr+8,addr(ISRold));
```

Nach Ende des Programms wird die Adresse der bisherigen ISR rekonstruiert. Ohne diesen Zugriff kann es durchaus passieren, daß der Rechner schlicht abstürzt.

```
  port[$21] := port[$21] or (1 shl IRQNr);
end.
```

Wieder der Zugriff auf das OCW1, falls zuvor der Interrupt gesperrt war, dann wird er nach getaner Arbeit auch wieder gesperrt. Wenn jetzt weiterhin Signale an den gesperrten IRQ gesendet werden, hat dies keinerlei Folgen mehr.

Derzeit teste ich diese Art der Interrupt-Programmierung mit einem Rechner MSM486SV, einem PC104-Board mit Prozessor Elan400 und 66 MHz. Kombiniert ist der Rechner mit einem AD-Board PC104-DAS08. Per Jumper ist auf diesem Board der IRQ5 freigegeben.

Auf dem AD-Board ist ein Flip-Flop, das mit ansteigender Flanke der Interrupt-Leitung gesetzt wird. Der Ausgang des Flip-Flop ist auf die Interrupt-Leitung des PC104-Bus (ISA-kompatibel) geführt.

Ein einlaufendes Signal setzt damit das Flip-Flop, dessen Ausgang den Interrupt auslöst. Die Interrupt-Service-Routine erhöht eine Zählvariable um 1 und setzt das Flip-Flop zurück. Da der IRQ5 vom Rechner nicht benutzt wird und somit auch keine ehemalige ISR bedient werden muß, ist die eigene ISR extrem schnell, so daß der Rechner bequem 100.000 Interrupts je Sekunde bedienen kann.

Eine sehr gute Zusammenstellung zu diesem Thema ist zu finden bei:

www.beyondlogic.org/interrupts/interupt.pdf