Hardware-Interrupts

Franz Fiala DSK-472:MYLIB.ZIP

Dieser Beitrag ist ein Abschnitt aus der Reihe „Hardwarenahes Programmieren in C“. Wie alle bisherigen Folgen ist auch diese Folge in sich abgeschlossen verwendbar. Alle folgenden Programmbeispiele sind auf der Begleitdiskette zu diesem Heft sowohl als C als auch als C++-Variante enthalten. Abgedruckt ist die C-Variante. C++ bringt, wenn es nur um die prinzipiellen Abläufe geht, keinen besonderen Vorteil. Erst wenn man daran geht, die Vorgänge rund um einen bestimmten Interrupt-Vektor in einer Klasse zu kapseln, kann man die Vorzüge von C++ nutzen. Diese Möglichkeiten werden in einem späteren Beitrag vorgestellt.

Für das Verständnis wäre es vermutlich einfacher, zuerst die leistungsfähigen Softwareinterrupts (BIOS, MSDOS-Interrupt 21h) vorzustellen, da Hardwarenähe aber im Vordergrund steht, wollen wir mit diesem schwierigeren Kapitel beginnen.

Stilfragen

Compiler brauchen keine sauberen Einrückungen und systematische Bennenungen, hauptsache die Syntax stimmt. Umso mehr brauchen es Leser von Programmen, um dem Verlauf folgen zu können. In meinen C-Programmen benutze ich die Regel, daß Typen, Konstanten und Makros groß geschrieben werden. Die Grundtypen `char, int, long` und ihre vorzeichenlosen Varianten `unsigned char..` werden in diesem Zusammenhang neu definiert und können wie `CHAR`, `INT`, `LONG`, `UCHAR`, `UINT`, `ULONG`... verwendet werden. Diese Festlegungen sind in einer eigenen Headerdatei `mytypes.h` festgehalten und müssen nicht in jedem kleinen Programm neu definiert werden.

Compilerhersteller bieten viel mehr an Funktionalität als in Standards festgelegt. Eine Flut praktischer Funktionen soll den Programmierer dazu verleiten, sie zu verwenden, statt darauf zu achten, ob sie auch in einer anderen Umgebung (anderer Compiler, andere CPU, andere Hardware, anderer Arbeitgeber?) verwendbar ist. Eine Unabhängigkeit dieser Art ist einmal die Unabhängigkeit von einem bestimmten Compiler. Egal, ob BORLAND, MICROSOFT, SYMANTEC oder WATCOM, portable Programme können das. Die Folge portabler Programmierung ist aber sehr zerhackter Kode. Ein unübersichtliche Folge von `#ifdef..#endif`-Zeilen ordnet diese Abhängigkeiten von Compiler, CPU und Hardware. Um diese immer weniger lesbaren Kodes wieder klarer zu machen, wird in den folgenden Programmen die Headerdatei `portable.h` angewendet, die viele Verschiedenheiten der einzelnen Compiler auf einen gemeinsamen Nenner zu bringen versucht.

Diese Problematik wird in den folgenden beiden Beispielen gezeigt.

Eine häufig verwendete Funktion ist das Löschen des Bildschirms. Während BORLAND-C (Ver. 3.1) dazu die Funktione `clrscr()` zur Verfügung stellt, bietet MICROSOFT-C (Ver. 8) dazu `_clearscreen(int)` an. Man sieht bereits, daß eine eindeutige Abbildung der Funktionen nicht möglich ist, denn der MICROSOFT-Compiler erwartet an dieser Stelle einen Parameter. In einem „zerhackten Kode“ würde das Bildschirmlöschen so aussehen:

```
#ifdef __TURBOC__
  clrscr();
#endif
#ifdef _MSC_VER
  _clearscreen(_GWINDOW);
#endif
```

Beschränken wir die Portabilität auf das einfache Bildschirmlöschen à la BORLAND, können wir in die Datei `portable.h` den folgenden Kode einfügen:

```
#ifdef _MSC_VER
  #define clrscr() _clearscreen(_GWINDOW)
#endif
```

Ganz ähnlich wird mit der BIOS-Funktion zur Tastaturkommunikation verfahren:

```
#ifdef _MSC_VER
  #define bioskey _bios_keybrd
#endif
```

Etwas diffiziler wird das Problem bei der BORLAND-Funktion `delay()`, die in Vielfachen von Millisekunden relativ fein abgestufte Zeitverzögerungen ermöglicht. Die Frage, wie wird’s gemacht wird, ist nicht ganz trivial, denn der eingebaute Timer (siehe **PCNEWS**-36, Seite 62..69) erlaubt nur Vielfache ab 50 ms.

Zeitverzögerungen im ms-Bereich

Aus einer sehr brauchbaren Darstellung der BIOS-Interrupts kann man entnehmen, daß der PC (ab dem AT) über eine wenig bekannte aber für Meßzwecke sehr praktische zusätzliche Echtzeituhr verfügt. Die Echtzeituhr wird über den Interrupt 15h, Funktionen 83h (ohne Wartefunktion) und 86h (mit Wartefunktion) eingeschaltet. Während sie läuft, generiert die Echtzeituhr nach jeweils 1024 µs einen Interrupt 70h. Man kann daher die von BORLAND gelieferte Funktion `delay()` folgendermaßen nachbilden:

```
void delay(unsigned ms)
  {
    union _REGS r;
    unsigned long us = ms * 1024L;
    r.h.ah = 0x86;
    r.h.al = 0;
    r.x.cx = (unsigned)(us/65536L);
    r.x.dx = (unsigned)(us%65536L);
    _int86(0x15,&r,&r);
  }
```

Portabilität

Der hier abgedruckte Programmkode ist ohne die Headerdatei `portable.h` weitgehend kompatibel mit der BORLAND-Schreibweise, da dieser Compiler an den Schulen überwiegend verwendet wird. Alle hier beschriebenen Programm sind aber ebenso mit dem MICROSOFT-Compiler lauffähig; alle, beim MICROSOFT-Compiler anderslautenden Namen sind durch entsprechende `#define` - Zeilen in `portable.h` substituiert. ACHTUNG: Wenn man mit mehreren Compilern arbeitet, merkt auch ohne Studium des ANSI- oder UNIX-Standard bald, welche Funktionen portierbar sind und welche nicht. Vermeiden wir Funktionen außerhalb von ANSI-C wo immer es geht und bereichern unsere Bibliothek mit selbstgeschriebenen, unabhängigen Funktionen.

Interrupts, Polling und DMA

Interrupts sind Hardwareereignisse, die den normalen Prozessorablauf unterbrechen können, um rasch auf ein externes Ereignis reagieren zu können.

Die Alternative zum Interrupt ist **Polling** (=Abfrage) der verschiedenen Quellen. Je länger diese Abfrageschleife ist, desto weniger kann eine bestimmte Reaktionszeit garantiert werden, je nachdem, an welcher Stelle sich die Polling-Routine gerade befindet. Bei einfachen Problemen kann Polling durchaus schneller als ein Interrupt sein, denn bei kurzer Pollingschleife (z.B. warten, bis ein Bit gesetzt ist) dauert der Overhead der Interruptprogrammierung unter Umständen länger. Bei komplexeren IO-Strukturen ist aber Interruptprogrammierung unbedingt erforderlich.

Bei großen zu übertragenden Datenmengen (Floppy, Festplatte, Netzwerk, Sound) wirkt der Datentransfer via CPU wie ein Flaschenhals und man geht zur **DMA**-Programmierung über, bei der die Daten direkt und ohne Mitwirkung der CPU vom und zum Speicher befördert werden. Allerdings ist dazu zusätzliche Hardware im PC (DMA-Controller) und auf der Peripherie-Baugruppe erforderlich.

Interrupt-Reaktion

Jeder Interrupt wird der CPU über eine Interruptleitung gemeldet. Je nach Aufbau der Prozessorhardware gibt es verschiedene Reaktionsmöglichkeiten auf Interrupts. Das Problem liegt darin, der CPU möglichst rasch mitzuteilen, wer etwas von ihr will. Die CPU verfügr ja nur über eine einzelne Leitung auf der sie angehalten wird.

Beim **Polling** (Serienabfrage aller in Frage kommenden Interruptquellen, nicht zu verwechseln mit einem Programm, das per Polling Hardware-Ereignisse feststellt) wird zwar der Interrupt über eine Interruptleitung gemeldet, doch muß der Prozessor der Reihe nach alle möglichen Interruptquellen abfragen, um festzustellen, welche von ihnen den Interrupt auslöste. Abgefragt werden Registerzustände (Flags).

Beim **vektorisierten Interrupt** (im PC verwendet), wird der Prozessor durch das Hardwareereignis sofort auf die richtige Adresse gesteuert.

Die Adresse der Interruptservice-Routine steht im **Interruptvektor**, der einen festgelegten Platz im Speicher hat. Es sind die Speicherplätze 00000..003ff (= 1024 Bytes), die auf jeweils 4 Bytes einen Interruptvektor enhalten. Es gibt daher 256 Interruptvektoren. Diese „Automatik“ spart Rechenzeit.

Ablauf eines Interrupts

Jeder Interrupt bewirkt, daß nach Abarbeitung des aktuellen Befehls die Adresse des Programmzählers auf den Stapel gelegt wird, um die geordnete Rückkehr von der Interruptservice-Routine zu ermöglichen. Insofern wäre kein Unterschied zu einem Unterprogramm gegeben. Darüberhinaus speichert aber ein Interrupt-Ereignis auch den Zustand der Flags am Stapel.

Ein Interruptservice-Routine (ISR), die durch einen Hardware-Interupt ausgelöst wurde, hat folgende Aufgaben zu erfüllen:

- Retten aller durch die Interruptroutine benötigten Register (*)
- Ausführen der diesem Interrupt zukommenden Aufgabe (Timer: Zeitzähler erhöhen, Tastatur: Tastenkode abholen und in Tastaturschlange ablegen..)
- Löschen der Interruptaufforderung (Rücksetzen des Interruptcontrollers)
- Wiederherstellen aller Register (*)
- Beenden mit einem IRET-Befehl, der im Gegensatz zu einem gewöhnlchen RET-Befehl auch die FLAGS wiederherstellt. (*)

Die mit (*) gekennzeichneten Aktivitäten übernimmt in einem C-Programm der Compiler.

Wie erfährt nun der Prozessor, auf welchen dieser 256 Interruptvektoren er schauen soll? Das besorgt im Falle des PC ein Interrupt-Controller (8259), der zu Beginn der PC-Existenz, durch das BIOS initialisiert wird. In dieser Phase wird ein Zusammenhang zwischen angeschlossener Hardware und den Interruptserviceroutinen hergestellt, indem die Interruptvektoren auf einen Anfangswert initialisiert werden. Nicht benötigte Interruptvektoren haben einen leeren IRET-Befehl als Ziel.

Ein erstes Programm `INTVEK.EXE`, dient zur Veranschaulichung der Interruptvekoren im PC-Speicher. Es liest 64 Interruptvektoren aus und stellt Ihre Adressen tabellarisch am Bildschirm dar. Das folgende Bild ist ein Beispiel für die Ausgabe:

```
Liste der Interrupt-Vektoren
============================
Aufruf: intvek      Die ersten 64 Vektoren
        intvek iii  Die Vektoren beginnend bei iii

 0   0 1917:  c0   1   1   70: 6f4   2   2  cf8:  16   3   3   70: 6f4
 4   4   70: 6f4   5   5 f000:ff54   6   6 f000:eb43   7   7 f000:eaeb
 8   8 154a:   0   9   9  f2b: 8d2   a  10  cf8:  57   b  11  cf8:  6f
 c  12 10dd:1ec3   d  13  cf8:  9f   e  14  cf8:  b7   f  15   70: 6f4
10  16 1555:   f  11  17 f000:f84d  12  18 f000:f841  13  19 e453:18c5
14  20 f000:e739  15  21 150c:   0  16  22 f000:e82e  17  23  55b: b34
18  24 f000:e000  19  25 e453:1990  1a  26 f000:fe6e  1b  27 cc29: 19f
1c  28 154a:  98  1d  29 f000:f0a4  1e  30    0: 522  1f  31 c000:5990
20  32  11c:1094  21  33 e453:16b4  22  34 180e: 2b1  23  35 180e: 14a
24  36 180e: 155  25  37 e453:19de  26  38 e453:1a27  27  39  11c:10bc
28  40 fef6: b82  29  41 cc29: 510  2a  42  55b: 59c  2b  43  11c:10da
2c  44  11c:10da  2d  45  11c:10da  2e  46  dbb: 13f  2f  47 1486: 424
30  48 1c10:d0ea  31  49 f000:ea01  32  50  11c:10da  33  51 10dd: e89
34  52  11c:10da  35  53  11c:10da  36  54  11c:10da  37  55  11c:10da
38  56  11c:10da  39  57  11c:10da  3a  58  11c:10da  3b  59  11c:10da
3c  60  11c:10da  3d  61  11c:10da  3e  62  11c:10da  3f  63  11c:10da
```

Man erkennt einzelne Vektoren, die mit dem Segment f000 beginnen. Diese Vektoren sind noch immer in jenem Zustand, der vom BIOS ursprünglich festgelegt wurde. Andere haben tiefliegende Adressen mit dem Segment 70 oder 11C. Das sind Vektoren, die auf die Programmteile zeigen, die beim Starten des Betriebssystems in den Dateien `IO.SYS` und `MSDOS.SYS` zuerst gewisse Änderungen an den ursprünglichen Vektoren vornehmen. Die höheren Nummern, etwa Segment e453 zeigen auf Treiber, die nach dem Systemstart geladen wurden und in die Upper Memory Blocks verschoben wurden.

Das Programm, das die obige Liste erstellt wurde in den **PCNEWS**-35, Seite 56..57 vorgestellt.

Interrupt-Arten

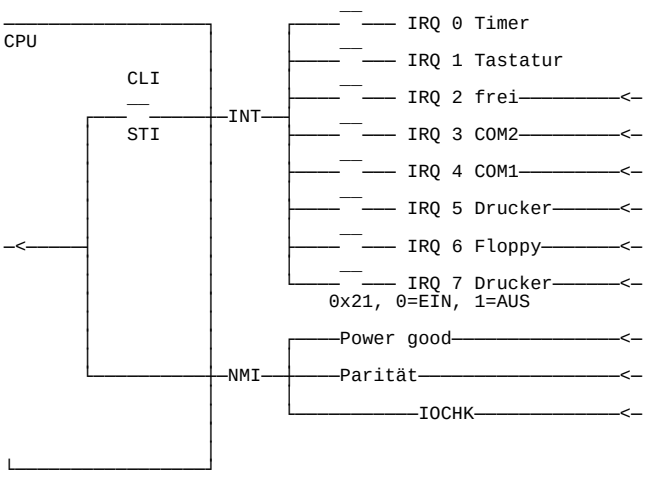
Hardware-Interrupts

Die ursprünglichen Interrupts (oder was man ursprünglich unter einem Interrupt versteht) sind die Hardware-Interrupts, ausgelöst durch externe Ereignisse, die Änderungen im Programmablauf bewirken. Ihr Funktionieren ist vom Konzept der Interruptvektoren abhängig aber auch vom Vorhandensein eines Interruptcontrollers, der erst den Vektor der CPU bekanntgibt.

Die CPU allein ist lediglich in der Lage einen Assemblerbefehl des Typs `INT n` auszuführen, der aus 2 Bytes besteht. das zweite Byte ist eine von 256 möglichen Interruptnummern. Dabei holt sich die CPU das Sprungziel von der Adresse n*4. Dieser Befehl wird während der BIOS-Initialisierung in den Interruptcontroller geschrieben und von diesem während der Interruptbearbeitung an die CPU weitergegeben. Die CPU führt also ausnahmsweise einen Befehl aus, der nicht aus dem Hauptspeicher kommt, sondern aus dem Interuptcontroller. Ganz allgemein könnte in diesem Zusammenhang der Interuptcontroller auch andere Befehle, etwa Sprungbefehle weiterleiten, er könnte auch mit anderen CPUs zusammenarbeiten.

Hardwareereignisse haben verschiedene Wichtigkeit. Man unterscheidet Ereignisse, die unter allen Umständen erkannt werden müssen und solchen, die abschaltbar sind. Die nicht abschaltbaren (oder nicht maskiebaren) Interrupts werden an eine eigene Prozessorleitung (NMI) geführt. Als Quellen dazu zählen Meldungen von der Stromversorgung oder Speicherfehler.

Alle anderen Interruptquellen werden über den Interruptcontroller geführt und enden letztlich an der Prozessorleitung IRQ. Im Interruptcontroller ist jede einzelne Quelle individuell ausschaltbar (maskierbar) und in der CPU können alle Interrupts gemeinsam durch den Befehl `CLI` aus- und mit dem Befehl `STI` eingeschaltet werden. Das folgende Bild gibt einen Überblick über diese Schaltmöglichkeiten.



Software-Interrupts

In der Systemprogrammierung ist es aber praktisch, wenn verschiedene Dienstleistungen des Betriebssystems immer auf denselben Adressen verfügbar sind. Daher wurde beim PC das Interruptkonzept insofern erweitet, als vielmehr Interruptvektoren vorhanden sind, als hardwarmäßig benötigt werden. Für die Hardware sind im AT 15 Interruptquellen möglich, die ebensoviele Interruptvektoren belegen. Alle anderen 241 Interuptvektoren können durch andere Programme so verwendet werden als wären sie Hardware-Interrupts, nur werden sie nicht durch Hardwareereignisse (asynchron zum Programmgeschehen) sondern durch die Programme selbst ausgelöst. Die erste Gruppe sind 16 BIOS-Interrupts, danach folgen 16 MSDOS-Interupts. Alle anderen werden durch spezielle Hardware benutzt oder sind für den Benutzer frei. Die Besprechung der Software-Interrupts erfolgt in einer der nächsten Folgen.

Interrupt-Nummern

Welcher Interrupt nun bei welchem Hardwareereignis ausgelöst wird, bestimmt die Programmierung des Interruptcontrollers. Festgelegt wurden die Nummern 8..15 für den ersten Interruptcontroller, der auch schon im XT existierte, und die Nummern A0..A7 für den zweiten Interruptcontroller.

Interrupts schalten, HC07IN1.C

Das individuelle ein- oder ausschalten der Interruptquellen erfolgt im Interruptcontroller. Die Adresse des ersten Interruptcontrollers ist 0x20 und 0x21. Während der Interruptcontroller über die Adresse 0x20 programmiert wird (normalerweise durch das BIOS), erfolgt das Ein- und Ausschalten der Interruptquellen über das Register auf Adresse 0x21. Jedem Bit des Interrupt-Enable-Registers von Adresse 0x21 entspricht ein Schalter in der obigen Skizze. Eine 1 schaltet den Interrupt aus, eine 0 schaltet ihn ein. Bei diesen Schalthandlungen muß man darauf achten, tatsächlich nur ein einzelnes Bit zu schalten und die anderen unverändert lassen. Man muß daher das Register zuerst lesen, danach das Bit setzen (oder löschen) und den gesamten Registerinhalt wieder zurückspeichern.

Das zeigen wir an einem Beispiel der Tastatur. 10 Sekunden lang gibt es keine Tastaturreaktion, während aber der Timer problemlos weiterläuft, Interruptleitung 0 ist davon nicht betroffen.

```
/* HC07IN1.C */
/*
 * Tastaturinterrupt ausschalten
 * =============================
 */
#include <dos.h>
#include <time.h>
#include <bios.h>
#include <stdio.h>
#include <conio.h>
#include <mytypes.h>
#define P_IO
#define P_BIOS
#define P_CONSOLE
#include <portable.h>
#ifdef _MSC_VER
   #include <graph.h>
#endif

VOID main(VOID)
{
  UCHAR inte;
  time_t zeit,endzeit;
  INT stop=0;

  clrscr();
  printf("Tastatur-Interrupt wird "
         "in den nächsten 10s ausgeschaltet\n"
        "Alle Tastendrucke sind unwirksam, "
         "ausprobieren!\n"
        "Start mit Taste, Ende mit ESC\n");
  putch(getch());

  time(&endzeit);
  endzeit += 10;

  // Tastatur-Interrupt ausschalten
  inte=IN_PORT(0x21);
  OUT_PORT(0x21,inte | 0x02);

  /* Tastaturpuffer räumen */
  for (;;)
  {
    UINT key;
    key=bioskey(1);
    if (key) // jeder Tastendruck wird dargestellt
    {
      printf("%4x ",bioskey(0));

      if (key==0x011b)  // ESC beendet
        break;
    }

    time(&zeit);

    if ((zeit>endzeit) && (stop==0))
    {
      // Tastatur-Interrupt einschalten
      inte=IN_PORT(0x21);
      OUT_PORT(0x21,inte & ~0x02);

      printf("\nJetzt müßte die Tastatur "
             "wieder funktionieren, Ende mit ESC\n");
      stop = 1;
    }
  }
  printf("\n\n");
}
```

Ausschalten des Timer-Interrupt

HC07IN2.C

Das funktioniert natürlich auch mit dem Timer.

```
/* HC07IN2.C */
/*
 * Ausschalten des Timer-Interrupts
 * ================================
 */
#include <dos.h>
#include <time.h>
#include <bios.h>
#include <stdio.h>
#include <conio.h>
#include <mytypes.h>
#define P_TIME
#define P_IO
#include <portable.h>

VOID main(VOID)
{
  UCHAR inte;
  time_t t;

  clrscr();
  printf("Testen des Ausschaltens "
         "des Timer-Interrupt\n"
         "ACHTUNG: läuft nicht in einem Windows-Fenster!\n");
  printf("Zunächst Timer normal,\n"
         "nach jeweils 1 s kommt "
         "der nächste Zählerstand: (Taste)\n");
  do
  {
    time(&t);
    printf ("%8lx ",t);
    delay(1000);
  }
  while(bioskey(1)==0);
  getch();

  printf("\nWarten Sie jetzt einige Sekunden "
         "und betätigen Sie dann wieder eine Taste:\n");
  getch();

  printf("Sie sehen, die Zeit läuft weiter.\n"
         "eine Taste dreht den Timer-Interrupt ab.\n");
  do
  {
    time(&t);
    printf ("%8lx ",t);
    delay(1000);
  }
  while(bioskey(1)==0); getch();

  printf("\nWarten Sie wieder einige Sekunden, "
         "dann noch einmal Taste\n");
  // Timer-Interrupt ausschalten
  inte=IN_PORT(0x21);
  OUT_PORT(0x21,inte | 0x01);
  getch();

  // Timer-Interrupt einschalten
  inte=IN_PORT(0x21);
  OUT_PORT(0x21,inte & ~0x01);
  printf("\nDie Zeit blieb stehen (Taste)\n");
  do
  {
    time(&t);
    printf ("%8lx ",t);
    delay(1000);
  }
  while(bioskey(1)==0);
  getch();
}
```

Interrupt mit Bibliotheksfunktion

HC07IN3.C

Wenn man mit Interruptprogrammierug häufig zu tun hat, ist es nützlich, die Schalthandlungen in zweckmäßigen Funktionen zusammenzufassen, wie im folgenden Beispiel in den Funktionen `int_on()` und `int_off()` gezeigt wird. Diese Funktionen sind im Programmodul `int.c` enthalten.

```
/* int.c */
#include <iodef.h>
```

```
#include <mylib.h>
#define P_IO
#include <portable.h>

VOID cdecl int_off(UCHAR nummer)
{
  UCHAR inte = IN_PORT(PC_PIC_ENABLE);
  OUT_PORT(PC_PIC_ENABLE, inte | (0x01<<nummer) );
}

VOID cdecl int_on(UCHAR nummer)
{
  UCHAR inte = IN_PORT(PC_PIC_ENABLE);
  OUT_PORT(PC_PIC_ENABLE, inte & ~(0x01<<nummer));
}
```

```
/* HC07IN1.C */
/*
 * Tastaturinterrupt ausschalten, mit Funktion
 * ===========================================
 */
#include <dos.h>
#include <time.h>
#include <bios.h>
#include <stdio.h>
#include <conio.h>
#include <mytypes.h>
#define P_IO
#include <portable.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\source\int.c"
#endif

VOID main(VOID)
{
  time_t zeit,endzeit;
  INT stop=0;

  clrscr();
  printf("Tastatur-Interrupt wird "
   "in den nächsten 10s ausgeschaltet\n"
         "Alle Tastendrucke sind unwirksam, "
   "ausprobieren!\n"
         "Start mit Taste, Ende mit ESC\n");
  getch();

  time(&endzeit);
  endzeit += 10;

  // Tastatur-Interrupt ausschalten
  int_off(1);

  /* Tastaturpuffer räumen */
  for (;;)
  {
    UINT key;

    key=bioskey(1);
    if (key) // jeder Tastendruck wird dargestellt
    {
      printf("%4x ",bioskey(0));

      if (key==0x011b)  // ESC beendet
        break;
    }

    time(&zeit);

    if ((zeit>endzeit) && (stop==0))
    {
      // Tastatur-Interrupt einschalten
      int_on(1);

      printf("\nJetzt müßte die Tastatur "
       "wieder funktionieren, Ende mit ESC\n");
      stop = 1;
    }
  }
  printf("\n\n");
}
```

Interrupt-Vektoren verändern, HC07IV1.C

Eine der wichtigsten Aktivitäten beim Umgang mit Interruptprogrammierung ist das Ersetzen, oder Erweitern bestehender Interruptservice-Routinen (ISR-Routine oder Interrupt-Handler). Eine neue Interrupt-Routine kann dabei

- **statt**
- **vor** oder
- **nach**

der alten Interrupt-Routine eingesetzt werden.

Da das ganze Betriebssystem vom richtigen Zusammenspiel der Interrupts abhängt, ist eine Veränderung der Interruptvektoren eine sehr absturzgefährdete Handlung. Es ist daher zweckmäßig, die grundlegende Arbeitsweise an Interupts zu probieren, die nicht zu den „lebenswichtigen" Interrupts im PC gehören. Wir begnügen uns zunächst damit, die alte Interrupt-Routine einfach durch die neue zu ersetzen.

Ein besonders angenehmer Kandidat zum Testen ist der Print-Screen-Interrupt, Nummer 5. Der Print-Screen-Interrupt wird zwar nicht direkt beim Drücken der Taste Print-Screen (Druck) sondern indirekt (zuerst wird die Taste durch den Tastatur-Handler abgeholt, die ihrerseits den Print-Screen-Interrupt auslöst), es hat aber dieselbe Wirkung als würde ein direkte Leitungsverbidung zwischen dieser Taste und dem Interruptcontroller bestehen.

Da jeder Interruptvektor 4 Bytes belegt (2 für Segment und 2 für Offset) finden wir den Print-Screen-Interrupt auf der absoluten Adresse 0x00014 (=20), als Segment und Offset angeschrieben auf Adresse 0x0000:0014.

Da der Vektor 4 Bytes belegt, können wir ihn als Variable vom Typ `long` betrachten. (Der Vektor ist keinesfalls eine Variable vom Typ `long`, er hat nur zufällig dieselbe Länge, und es wird später ein besseres Verfahren vorgestellt, dem Vektor zu Leibe zu rücken.) Da die Vektoradresse eine absolute Adresse ist, benutzen wird das Makro `MK_FP()`, um den Vektor zu initialisieren und auf die richtige Adresse zeigen zu lassen. Bevor wir das Programm starten, empfiehlt es sich, mit dem vorhin vorgestellten Hilfsprogramm `INTVEK` den aktuellen Wert auszulesen, um die Richtigkeit des folgenden Programms zu prüfen.

Das Programm `HC07IV1.C` liest den Vektor aus, setzt ihn auf einen neuen Wert (0x12345678), zeigt auch diese Wert an und stellt schließlich wieder den alten Wert ein.

```
/* HC07IV1.C */
/*
 * Interruptvektor ersetzen
 * ========================
 */
#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <mytypes.h>
#define P_INT
#include <portable.h>

VOID main(VOID)
{
  ULONG far *ip;
  ULONG alter_vektor;

  clrscr();
  ip=(ULONG far *)MK_FP(0,20);
  disable(); alter_vektor = *ip; enable();
  printf
  ("Geretteter PrintScreen-Interrupt-Vektor: %lFp\n",
    alter_vektor);
  disable(); *ip = 0x12345678L; enable();
  printf("Neuer PrintScreen-Interrupt-Vektor: %lFp\n",
          *ip);
  disable(); *ip = alter_vektor; enable();
  printf("Wiederhergestellter alter Vektor: %lFp\n",
    *ip);
  getch();
}
```

Sie sehen, daß man für das Handling mit den Interrupt-Vektoren keine besondere Funktion benötigt, Pointer genügen. Da aber ein Interruptvektor auch einen besonderen Typ hat, wird im Beispiel `HC07IV2A.C` eine typenrichtige Lösung vorgestellt.

Interrupt-Service-Routinen, HC07IV2.C

Um in C eine Interruptservice-Routine schreiben zu können, wurde in der Microsoft-Variante von C ein neues Schlüsselwort `interrupt` eingeführt, das praktisch von allen anderen Herstellern übernommen wurde. Eine gewöhnliche Funktion erfüllt ja nicht die Bedingung, daß Register gerettet (und vor Beendigung wiederhergestellt) werden müssen und der Befehl `IRET` statt `RET` die Funktion beschließt. Dieses neue

Schlüsselwort verleiht einerseits der Routine einen einen neuen Typ und auch die erforderlichen Features.

Was man in einer Interrupt-Service-Routine ohne zusätzliche Maßnahmen tun darf und soll, ist beschränkt. Man kann eigene Funktionen rufen, Variablen verändern. Betriebssystem- oder BIOS-Funktionen dürfen weder direkt noch indirekt gerufen werden. Wir müssen ja bedenken, daß Hardware-Interrupts zu jedem beliebigen Zeitpunkt eintreffen können und daß keinerlei Voraussagen getroffen werden können, in welchem Zustand oder in welchem Programm sich das Betriebssystem gerade befindet.

Merken wir uns daher als wichtigste Regel, daß man in Interrupt-Service-Routinen keine Funktionen rufen darf, die ihrerseits in irgendeiner Form das Betriebssystem rufen können. (Bildausgabe, Dateienoperationen usw.).

Unser erstes Beispiel beschränkt sich daher auf die Veränderung einer Bildschirmadresse (Zeichen in der linken oberen Ecke).

Wir verwenden noch immer die Technik des vorigen Beispiels, den Interrupt-Vektor als eine Variable vom Typ `long` zu betrachten.

Beachten Sie auch, daß die ursprüngliche Funktion der Print-Screen-Taste völlig brachliegt, die Routine wird nicht weiter aufgerufen. Der neue Handler arbeitet *statt* dem alten.

```
/* HC07IV2.C */
/*
 * Interrupt-Service-Routine
 * =========================
 */
#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <mytypes.h>
#define P_INT
#include <portable.h>

VOID interrupt far prt_scr(VOID)
{
  UCHAR far *bild=(UCHAR far *)MK_FP(0xb800,0);
  (*bild)++;
}

VOID main(VOID)
{
  ULONG far *ip;
  struct SREGS sr;
  ULONG alter_vektor;

  clrscr();
  printf("Umlenkung eines Interruptvektors "
         "auf das eigene Programm\n");
  ip=(ULONG far *)MK_FP(0,20);
  printf("Alter PrintScreen-Interrupt-Vektor: %lFp\n",
         *ip);

  disable(); alter_vektor = *ip; enable();

  /* Umlenken des PrintScreen-Interrupt */
  segread(&sr);
  disable();
  *ip=(ULONG)MK_FP(sr.cs,(UINT)prt_scr);
  enable();

  printf("Neuer PrintScreen-Interrupt-Vektor: %lFp\n",
         *ip);
  printf("Die neue Interrupt-Service-Routine "
         "verändert den Inhalt\n"
         "der Bildschirmzelle 0,0 links oben "
         "mit PRTSCR\n"
         "Jede andere Taste "
         "stellt den alten Vektor wieder her\n");

  do {} while (!kbhit()); getch();

  disable(); *ip = alter_vektor; enable();
  printf("Wiederhergestellter alter Vektor: %lFp\n",
         *ip);
  getch();
}
```

Bibliotheksfunktionen für Interruptbearbeitung HC07IV2A.C

In der Bibliothek des C-Compilers sind bereits Funktionen zur Handhabung von Interrupt-Vektoren enthalten. Sie heißen `getvect()` und `setvect()`. Als Argumente bzw. Rückgabewerte verlangen Sie Zeiger auf Interruptserviceroutinen. Ein solcher Zeiger `vector` hat den Aufbau

```
void interupt (*vektor) ();
```

```
/* HC07IV2A.C */
/*
 * Interrupt-Service-Routine
 * =========================
 */
#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <mytypes.h>
#define P_INT
#include <portable.h>

VOID interrupt prt_scr()
{
  UCHAR far *bild=(UCHAR far *)MK_FP(0xb800,0);
  (*bild)++;
}

VOID main()
{
  ULONG far *ip;
  struct SREGS sr;
  VOID (interrupt *alter_vektor)();

  clrscr();
  printf("Umlenkung eines Interruptvektors\n");

  ip=(ULONG far *)MK_FP(0,20);
  printf("Alter PrintScreen-Interrupt-Vektor: %lFp\n",
         *ip);

  alter_vektor = getvect(5);

  /* Umlenken des PrintScreen-Interrupt */
  segread(&sr);

  setvect(5,prt_scr);

  printf("Neuer PrintScreen-Interrupt-Vektor: %lFp\n",*ip);
  printf("Die neue Interrupt-Service-Routine "
         "verändert den Inhalt\n"
         "der Bildschirmzelle 0,0 links oben "
         "mit PRTSCR\n"
         "Jede andere Taste "
         "stellt den alten Vektor wieder her\n");

  do {} while (!kbhit()); getch();

  setvect(5,alter_vektor);

  printf("Wiederhergestellter alter Vektor: %lFp\n",
  *ip);
  getch();
}
```

Funktionen im Interrupt-Handler

HC07IV3.C

Daß man keine Funktionen in der Interrupt-Service-Routine verwenden darf, kann man leicht probieren. Man schreibe statt der Pointer-Manipulation des Bildschirms den Befehl `printf(„A“)` in die Interruptservice-Routine und starte das Programm. Genau einmal wird es (vielleicht) funktionieren, dann hilft nur mehr CTRL-ALT-DEL. Dafür sind mehrere Gründe zuständig:

DOS- und BIOS-Funktionen sind nicht „reentrant“,

d.h. innerhalb eines MSDOS-Aufrufs darf im allgemeinen nicht gleichzeitig wieder ein MSDOS-Aufruf erfolgen. MSDOS-Funktionen benutzen globale Variablensätze und würden daher bei nochmaligem Aufruf diese Variablen überschreiben. Vergleich: Funktionen in C sind durchaus „reentrant“, da sie rekursiv aufrufbar sind, wenn man nur dafür sorgt, daß sie nur lokale Variablen benutzen, denn diese werden für jeden Aufruf neu am Stapel angelegt. Diese Eigenschaft, rekursive Funktionen schreiben zu können, wird aber nur fallweise in C-Funktionen ausgenutzt.

Darf ich herein?

Da Hardware-Interrupts völlig asynchron zum Programmgeschehen eintreffen, werden sie immer wieder auch dann auftreten, wenn das Betriebssystem gerade etwas zu tun hat. MSDOS meldet mit dem sogenannten `INDOS`-Flag, ob ein Betriebssystemaufruf erfolgen kann. Dieses Flag war lange Zeit nicht offiziell dokumentiert, wurde aber immer für

diesen Zweck immer verwendet. Zum ersten Mal offiziell erwähnt wird es im „MS-DOS Programmer's Reference", ISBN 1-55615-329-5.

Stackbeschränkung

Auch wenn man sie vielleicht unterbrechen könnte, verwenden DOS-Funktionen einen internen Stack mit sehr beschränkter Länge (128 Bytes). Jede unvorhergesehene Belastung dieses Stack kann zum Überlauf führen. Speziell komplexe Unterprogramme mit Parameterübergabe wie z.B. `printf()` sind davon betroffen. Daher sind Interrupt-Routinen gut beraten, für die Dauer ihrer Tätigkeit einen eigenen Stack aufzubauen und nach Gebrauch wieder den alten einzusetzen.

Weitere Probleme gibt es bei Diskettenoperationen und residenten Programmen, die hier nicht weiter behandelt werden. Es wird dazu einen eigenen Abschnitt „residente Programme" geben.

Diese Probleme können nun mit den Mitteln von C allein nicht mehr bewältigt werden. Es gibt Compiler, die Lösungen für diese Problematik mitliefern (SYMANTEC-C++) oder Bibliotheken, die mit eigenen Lösungen aufwarten (TSR and More von Turbo-Power).

Das folgende Programm zeigt einen Lösungsansatz für BORLAND-C, indem zumindest das Stack-Problem durch kurze Assembler-Routinen am Beginn und am Ende der Interrupt-Service-Routine Absturzsicherheit bieten. Das Programm ist für Experimente gedacht und sollte durch eine Abfrage des INDOS-Flag erweitert werden.

Jedes Drücken der Print-Screen-Taste bewirkt das Hochzählen eines Zählers und die Anzeige mit `printf`-Befehl.

```
/* HC07IV3.C */
/*
 * Funktionen in der Interrupt-Service-Routine
 * ===========================================
 */
#if (__TURBOC__ && __TINY__) || (_MSC_VER && _M_I86TM)

#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <mytypes.h>
#define P_INT
#include <portable.h>

#ifdef __cplusplus
const UINT STACKLEN=5000;
#else
#define STACKLEN 5000
#endif

static CHAR temporary_stack[STACKLEN];
static UINT old_stackp;
static UINT old_stacks;

static UINT counter=0;

VOID interrupt far prt_scr()
{
  asm {
    cli
    // make TINY conditions cs=ds=es
    mov ax,cs
    mov es,ax
    mov ds,ax
    // remind old_stack
    mov old_stackp,sp
    mov ax,ss
    mov old_stacks,ax
    // make new_stack
    mov ax,cs
    mov ss,ax
    mov sp,OFFSET temporary_stack
    add sp,STACKLEN-1
    sti
      }
  counter++;
  cprintf("COUNTER:%i\n\r",counter);
  asm {
    // get old_stack
    cli
    mov ax,old_stackp
    mov bx,old_stacks
    mov ss,bx
    mov sp,ax
    sti
      }
}

VOID main(VOID)
{
  struct SREGS sr;
  VOID (interrupt far *alter_vektor)();

  clrscr();
#if __TURBOC__
  directvideo=1;
#endif
  printf("Benutzung h"herwertiger Funktionen "
         "in der Interrupt-Service-Routine\n");
  segread(&sr);
  if (sr.cs!=sr.ds)
  {
    printf("TINY-Modell verwenden!\n");
    getch();
    return;
  }

  alter_vektor=getvect(5);

  setvect(5,prt_scr);
  printf("Vektor umgelenkt\n");

  do {} while (!kbhit());

  setvect(5,alter_vektor);
  getch();
}

#else
  #error Must use TINY model
#endif // __TINY__
```

Komfortable Bibliotheken HC07IV3A.C

Viel bequemer ist es, wenn der Compiler bereits für diese Programmiertechnik ausgerüstet ist wie z.B. der SYMANTEC-Compiler (Früher ZORTECH). In der Zeile `#ifdef __ZTC__` werden Kompilierungen mit anderen Compilern verhindert. Die Funktionen `int_intercept()` und `int_restore()` stammen aus der Bibliothek dieses Compilers und berücksichtigen alle Randbedingungen beim Umgang mit Interrupts. Aus Platzgründen ist dieses Programm nicht abgedruckt aber auf der Programmdiskette zu diesem Heft enthalten.

Timer-Interrupt umlenken HC07IV4.C

Diese Einschränkungen bei der Benutzung von Funktionen in Interrupt-Service-Routinen sind nicht immer gravierend. Denn eine Interruptroutine soll ja gar keine zeitaufwendigen Aufgaben übernehmen. Sie soll so kurz wie möglich auf das Ereignis reagieren, dem Hauptprogramm Hinweise geben und sich so rasch wie möglich wieder verabschieden, um dem Hauptprogramm möglichst wenig Zeit zu stehlen.

Wie kann man es aber lösen, wenn man jedes Interruptereignis z.B. am Bildschirm mitprotokollieren will?

Die normale Kommunikation zwischen Interrupt-Service-Routine und Hauptprogramm sind eigene Variablen („Flags"), die dem Hauptprogramm das Ereignis melden. Das Hauptprogramm kann seinerseits der Interrupt-Service-Routine über dasselbe Flag zurückmelden, daß eine Reaktion erfolgt ist.

Zum ersten Mal wollen wir einen wichtigen Interrupt, den Timer-Interrupt verändern. Dabei müssen wir beachten, daß wir uns in eine bestehende Interrupt-Verarbeitungkette „einklinken", d.h. Programme vor uns (Treiber) und möglicherweise auch nach unserer Aktivität diesen Interrupt mit benutzen wollen. Der neue Timer-Interrupt-Handler schließt also nicht einfach sang und klanglos wie beim Print-Screen-Interrupt den Interrupt ab, sondern beendet erst nachdem er den vorigen Interrupt-Handler aufgerufen hat.

Der Timer-Interrupt wird 18-mal pro Sekunde ausgelöst. Seine wichtigste Ausgabe besteht darin, eine Variable vom Typ `long` im BIOS-Variablensegment zu inkrementieren. Außerdem bietet er allen Programmen seine Dienste an, die den Eindruck einer Gleichzeitigkeit von Abläufen erwecken wollen („Hintergrundaktivitäten").

Das Hauptprogramm erfährt von der Hintergrundaktivität über die Variable `count` und stellt ihren Wert am Bildschirm dar. Diese Technik beantwortet auch die Frage, wie man Bildschirmausgaben in Interrupt-Routinen durchführen kann. Die Antwort ist: nich in der Interruptroutine selbst, sondern im Hauptprogramm. Die Kommunikation zwischen

beiden übernimmt ein Flag, wobei auch das Hauptprogramm dieses Flag (je nach Aufgabenstellung) wieder zurücksetzen kann.

Man beachte das Schlüsselwort `volatile`, das der Variablen `count` vorangestellt wurde. Es bedeutet, daß sich diese Variable jederzeit ändern kann, und daß der Compiler diese Variable bei jeder Berechnung neu aus dem Speicher holt und nicht etwa eine bereits in den Registern befindliche Kopie benutzt.

```
/* HC07IV4.C */
/*
 * Timer-Interrupt umlenken
 * ========================
 */
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <mytypes.h>
#define P_INT
#define P_CONSOLE
#include <portable.h>

#define INTR 0x1C    /* TIMER-Interrupt */

VOID (interrupt far *oldhandler)();

volatile INT count=0;

VOID interrupt far handler()
{
   count++;         /* Zähler weiterzählen */
}

VOID main(VOID)
{
   INT oldcount=0;

   clrscr();
   printf("Timer-Interrupt zählt die Variable count hoch\n"
    "Start mit Taste\n");
   getch();

   /* alten Vektor merken */
   oldhandler = getvect(INTR);
   /* neuen Vektor installieren */
   setvect(INTR, handler);
   /* zählen und warten */
   while (count < 20)
     if (count!=oldcount)
     {
       printf("%d ",count);
       oldcount=count;
     }
   /* alten Vektor wiederherstellen */
   setvect(INTR, oldhandler);
   printf("\n");
   getch();
}
```

Tastatur-Interrupt umlenken HC07IV5.C

Das folgende Programm verwandelt alle Eingaben, die in Großbuchstaben erfolgen in Kleinbuchstaben und umgekehrt.

Um dieses Programm zu verstehen, ist es notwendig, sich die BIOS-Variablen rund um die Tastatur genauer anzusehen.

Jede Tastenberührung an der Tastatur löst am PC den Tastatur-Interrupt mit der Nummer 9 aus. Die Aufgabe dieses Interrupts ist es, den Tastencode zu identifizieren, gegebenenfalls auf ein spezielles Tastaturlayout (GR, IT, FR, US..) zu achten und das Ergebnis in 2 Bytes in einem zirkularen Buffer im BIOS-Datensegment abzulegen und dort für den Anwendungsprogrammierer (und den abholenden Interrupt 0x16) bereit zu stellen. Der Tastatur-Puffer ist 16 Zeichen groß. Wird die Taste nicht abgeholt, wird ein Zeiger auf die nächste freie Position gesetzt, solange, bis der Puffer voll ist; dieser Zustand wird mit einem Piepsen des Lautsprechers angekündigt. Auf Adresse 0x0040:0x001a befindet sich ein Zeiger, der auf die Stelle zeigt, auf der sich die letzte Tastatureingabe befindet, diese Stelle ändert sich von Tastendruck zu Tastendruck.

Unser Programm bedient sich dieser Zusammenhänge und konvertiert das Zeichen im Tastaturpuffer, wenn es ein Buchstabe ist `if ((*kp & 0x0060) == 0x0060) ...else if ((*kp & 0x0040) == x0040)`. Anders als beim Timer-Interrupt wird die alte Interrupt-Service-Routine *vor* der Modifikation ausgeführt, denn der alte Handler sorgt dafür, daß der Tasten- und ASCII-Kode im Buffer abgelegt wird.

```
/* HC07IV5.C */
/*
 * Tastatur-Interrupt umlenken
 * ===========================
 */
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <mytypes.h>
#define P_INT
#include <portable.h>

#define INTR 0x9    /* Tastatur-Interrupt */

VOID (interrupt far *oldhandler)();

VOID interrupt far handler()
{
   UINT far * kp;
   oldhandler();    /* alten Interrupt rufen */
   disable();       /* Interrupts ausschalten */
   kp = (UINT far *)MK_FP(0x0040,0x001a);
   kp = (UINT far *)MK_FP(0x0040,*kp);
   if ((*kp & 0x0060) == 0x0060)
     *kp = (*kp) & ~0x0020;
   else if ((*kp & 0x0040) == 0x0040)
     *kp = (*kp) | 0x0020;
   enable();        /* Interrupts einschalten */
}

VOID TastenTest(VOID)
{
   INT c;
   printf("\nTastaturecho, Ende mit CTRL-C\n");
   do
   {
     c=getch();
     printf("%c",c);
   }
   while ((c)!=3);
   printf("\n");
}

VOID main(VOID)
{
   clrscr();
   printf("Testen Sie Ihre Tastatur!\n");
   TastenTest();

   printf("Neue Interrupt-Service-Routine "
    "vertauscht Groß- und Kleinbuchstaben\n");

   /* alten Vektor merken */
   oldhandler = getvect(INTR);
   /* neuen Vektor installieren */
   setvect(INTR, handler);
   TastenTest();

   /* alten Vektor wiederherstellen */
   setvect(INTR, oldhandler);
   getch();
}
```

Der Timer läuft schneller, HC07IT0.C

Die normale Periode zur Unterbrechung eines Vordergrundprogramms ist 53 ms (18 mal pro Sekunde). Für normale Anwendungen ist dieser Wert klein genug, um unbemerkt eine Hintergrund-Aufgabe oder eine quasi-gleichzeitig stattfindende auszuführen. Anderseits ist sie auch groß genug, um ein Vordergrundprogramm nicht unnötig oft zu verzögern.

Es gibt aber Anwendungen, z.B. bei der Übertragung über die parallele Schnittstelle in festen Zeitabständen oder bei Spielen oder bei abtastenden Messungen, wo dieser zeitiche Abstand einfach zu groß ist, um beim Benutzer akzeptables Verhalten zu erreichen. Immer, wenn wir ein schnelleres Zeitmaß benötigen, ist es erforderlich, den Timer-Kanal-0 umzuprogrammieren.

Der Haken dabei ist, daß jede Beschleunigung des Timers auch eine Beschleunigung der internen Uhr zur Folge hat, da die interne Uhr vom Timer-Kanal-0 abgeleitet wird. Um hier fehlerfreie Programme zu erreichen gibt es zwei Methoden:

a. die „brutale“: Die interne Uhr ist uns gleichgültig, nach Beendigung unseres Programms holen wir die Zeit aus dem CMOS-RAM und setzen die Uhrzeit neu.

b. die „sanfte“: Wir berücksichtigen die schneller laufende Uhr und rufen die Timer-Interrupt-Routine entsprechend seltener auf.

Es ist klar, daß die 'brutale' Methode nur bedingt funktioniert. Was etwa, wenn das Programm Dateien speichert, dann bekommen diese alle eine falsche Uhrzeit verpaßt! Bei der Interrupt-Programmierung muß man sich bewußt sein, daß man nicht allein ist auf der Welt und im PC, sondern, daß sich viele andere Programme auch an den Timer-Interrupt anhängen und wir erstaunt sein werden, was dann alles nicht mehr funktioniert.

Können wir diesen beschleunigten Timer beliebig schnell machen? Wo sind die Grenzen?

Betrachten wir die 'sanfte' Methode, dann können wir noch zwei Variationen unterscheiden:

1. die 'brutal-sanfte': Wir erhalten einen beschleunigten Timer-Interrupt, kümmern uns aber nicht um etwaige andere Hintergrundaufgaben, sondern zählen in Eigenregie nur die Zeit im Hauptspeicher hoch, sodaß der Fehler der 'brutalen' Methode nicht aufscheint, dafür können wir vermutlich die Beschleunigung sehr hoch treiben, da niemand anders unsere Timer-Routine ungewollt verlängert.

2. die 'wirklich sanfte': Wir lassen alle 53 ms die alte Interrupt-Routine 'zu Wort kommen'. Das ist zwar vornehm, aber bedeutet für uns eine Einschränkung: Die Zeit, die die alte Interruptroutine gemeinsam mit unserer neuen Aufgabe braucht beschränkt die Möglichkeit zur Beschleunigung des Timers.

Zuerst wollen wir ganz sorgsam verfahren und demonstrativ zeigen, wie sich die Timer-Beschleunigung auswirkt und wo ihre Grenze ist. Die Beschleunigung geht so:

Der Anfangszählerstand (nach dem Booten) des Timers wird, ausgehend vom Anfangswert 0 (=10000h) immer halbiert, und dabei gleichzeitig eine globale Variable `divisor`, ausgehend vom Wert 1 verdoppelt. Der neue Timer-Interrupt-Handler, zählt eine Variable `count` solange hoch, bis `divisor` erreicht ist, dann aktiviert er den alten Timer-Interrupt.

Wichtige Voraussetzungen: Es darf nicht der Benutzer-Timer-Interrupt 1ch, sondern es muß der Timer-Interrupt 8h verwendet werden, da im Falle von 1ch die eigentliche Timeraufgabe, das Hochzählen der Zeit, bereits ausgeführt worden wäre. Da wir aber jetzt die ersten sind, die den Timer-Interrupt bearbeiten (und nur im Ausnahmefall die alte Routine aufrufen) sind wir auch dafür verantwortlich, den sogenannten EOI-Kode (0x20) an den Interrupt-Kontroller zu schicken, der (zufällig) auf die Adresse 0x20 zu schreiben ist. Dann aber gehts!

Als Indikator für die Beschleunigung verwenden wir der Einfachheit halber, wie so oft, den eingebauten Lautsprecher, sie können aber auch einen zusätzlichen Lautsprecher an die Datenleitung D0 des Druckerports LPT1 anschließen (Achtung: Vorwiderstand, ca. 200 Ohm nicht vergessen!).

```
/* HC07IT0.C */
/*
 * Variable Frequenzen durch Timer-Interrupt
 * =========================================
 * nicht den Benutzer-Timer-Interrupt 1Ch,
 * sondern 08h benutzen!
 */
#include <stdio.h>
#include <conio.h>
#include <bios.h>
#include <dos.h>
#define P_INT
#define P_IO
#include <portable.h>
//#include <mylib.h>
#include "..\source\tic.c"
#include "..\source\spk.c"
#include "..\source\tim.c"

#define INTR 0x08    /* TIMER-Interrupt */
VOID (interrupt far cdecl *oldhandler)();
VOID frequency(ULONG c, UINT div);

INT divisor = 1;
volatile count=0;

VOID interrupt timer_handler()
{
   spk_toggle();
   // PAR_TOGGLE();
   if (++count==divisor)
   {
     count=0;
     oldhandler();    /* alten Interrupt rufen */
   }
   else
   {
     OUT_PORT(0x20,0x20);
   }
}

VOID main(VOID)
{
   oldhandler = getvect(INTR); /* alten Vektor merken */

   clrscr();
   printf("Timerintervall von 53ms "
          "schrittweise verkleinern\n");
   printf("Kontrolle mit Lautsprecher, "
          "weiter jeweils mit Taste\n");
   printf("!! Nicht mit CTRL-BREAK unterbrechen !!\n");
   getch();

   disable(); setvect(INTR, timer_handler); enable();
   frequency(0x10000L,1);
   frequency(0x8000L,2);
   frequency(0x4000L,4);
   frequency(0x2000L,8);
   frequency(0x1000L,16);
   frequency(0x0800L,32);
   frequency(0x0400L,64);
   frequency(0x0200L,128);
   frequency(0x0100L,256);
   frequency(0x0080L,512);
   disable(); setvect(INTR, oldhandler); enable();
}

VOID frequency(ULONG c, UINT div)
{
  FLOAT f = 4770000.0F/(4*c);
  printf("\nFrequenz: %6.2f Hz "
         "Periodendauer: %6.2f us\n",f,1000000L/f);
  disable();
  divisor=div;
  tim0_set((UINT)c);
  enable();
  do { tic_show(); } while (bioskey(1)==0); bioskey(0);
}
```

Fragen: Bei welcher Timer-Beschleunigung wird bei Ihrem Rechner der Ton unsauber? Nie? Gratulation, Sie haben 'ein schnelles Eisen'! Was müßte man tun, um beispielsweise eine Verfünffachung (und nicht eine Vervierfachung) zu erreichen? Schreiben Sie eine allgemeine Funktion, die unsere Aufgabe für eine beliebig wählbare Zeit erfüllt und testen Sie diese!

Anmerkung: Der Autor hatte beim Testen dieses Programms Schwierigkeiten mit den Gleitkommaeinstellungen von BORLANDC-2.0. Es ging nur mit 'Fast-floating-point OFF' und keine Emulation sondern nur mit '80287'. Andere Einstellungen führten zu nicht näher erkärten Speicherfehler und Abstürzen, ohne auch nur die erste Zeile auszugeben, in der eine Ausgabe einer Gleitkommazahl gewünscht war. Erklärungen erwünscht!

Wie man dieses Prinzip nutzbringend anweden kann? Nehmen wir an wir hätten irgendeine Programmieraufgabe und wollten im Hintergrund etwas gleichzeitig erledigen. Die Programmieraufgabe ist einfach: die Eingabe eines Textes; die Hintergrundaufgabe ist das Bewegen einer Figur am Bildschirm und die Ausgabe eines Tones. Innerhalb des Textausgabeprogramms können die Hintergrundprogramme ein- und ausgeschaltet werden sowie ihre Geschwindigkeit verändert werden:

Hintergrundprogrammierung,

HC07IT1A.C, HC07IT1.C

Programme, die wesentliche Interrupts, wie den Timer-Interrupt einer ist, verändern, sind extrem absturzgefährdet, da am Timer0 die Existenz vieler residenter Programme aufbaut. Es ist daher ratsam, die Hintergrundaufgabe, die für das Interruptprogramm nur ein kurzer 'Durchlaufer' ist, vorher als Hauptprogramm zu testen. In unserem Fall ist das Ein- und Ausschalten des Lautsprechers schon getestet worden, es bleibt nur die Bedienung des Bildschirms:

Das Programm malt ein Zeichen auf den Bildschirm, hier das Zeichen 0x02 ('Gesicht') und bewegt es im Winkel von 45 Grad solange, bis es an einem der Ränder ankommt. Dann wird das Gesicht reflektiert, die Bewegungsrichtung kehrt um.

Während das Programm `HC07IT1A.C` (hier nicht abgedruckt) ein reines Vordergrundprogramm ist, das nur dazu dient, den Bewegungsablauf des Virus zu

testen (die Verzögerung besorgt die Funktion `delay()`); ist das Programm `HC07IT1.C` die endgültige Version, bei der der Virus durch die Interruptroutine gesteuert wird und das Vordergrundprogramm den Virus beeinflußt (die Ablaufgeschwindigkeit ist durch den Timerinterrupt gegeben).

Statt der Verzögerung durch den Timer finden Sie hier: `delay(100)`. Die Variable `fun` schaltet die Bildschirmbewegung ein und aus. Beachten Sie, daß das Programm nur auf CGA, EGA oder VGA, nicht aber auf MDA oder HGC läuft; in diesen Fällen wäre die Segmentadresse des Bildanfangs von `0xb800` auf `0xb000` zu ändern. Die Variablen `x` und `y` sind die augenblicklichen Koordinaten des Bewegungsablaufes, `dx` und `dy` sind die Bewegungszu- bzw. -abnahme. Es beginnt mit einer Bewegung in positive x und y-Richtung mit `dx=1` und `dy=1`. Die Umkehr der Bewegungsrichtung erfolgt, wenn die Figur an den Bildschirmrändern anlangt, `dx` und `dy` werden dann `-1`. Damit unser Bildschirmvirus das alte Bild wiederherstellen kann (es ist ein sympatischer Virus) merkt sich die Interrupt-Routine in der Variablen `fun_old_char` den gerade überschriebenen Wert. Die Variable `fun_start` ist zu Beginn auf `1` gesetzt und verhindert, daß der Beginnpunkt mit einem falschen Zeichen überschrieben wird, sie wird gleich nach dem ersten Durchlauf auf 0 gesetzt. Ebenso ermöglich sie es, daß - nach dem Beendigen des Hintergrundprogramms - die letzte überschriebene Bildschirmstelle widerhergestellt wird und nicht etwa die letzte Virusposition bestehen bleibt. Schließlich haben wir einen Zeiger `fp` auf den aktuellen Bildschirmpunkt. Die Variablen, die durch den Interrupt-Handler beeinflußt werden sind `volatile`, das signalisiert dem Compiler, daß sich ihr Wert immer ändern kann.

Das Programm selbst ist einfach. An den Rändern erfolgt eine Umkehr der Bewegungsrichtung. Die Taste beendet das Programm nicht sofort, sondern setzt zuerst die Hintergrundprogramme in den Aus-Zustand, wartet mit `delay(100)` noch einen Timer-Interrupt ab, damit die beschriebene Löschung des Virus möglich wird und beendet das Programm.

Erst wenn dieses Programm ausreichend getestet wurde, können wir daran gehen, es auch in die Timer-Interrupt-Service-Routine einzubauen:

```
/* HC07IT1.C */
/*
 * Texteingabe mit Hintergrundprogramm
 * ===================================
 */
#include <stdio.h>
#include <conio.h>
#include <bios.h>
#include <dos.h>
#define P_INT
#define P_IO
#include <portable.h>
#include "..\source\tim.c"
#include "..\source\spk.c"

VOID tim0_div(VOID);

#define INTR 0x08    /* TIMER-Interrupt */

VOID (interrupt far cdecl *oldhandler)();

INT divi = 1;
volatile count=0;
volatile INT spk = 1;
volatile INT fun = 1;
volatile INT x=78, y=7;
volatile INT dx=1, dy=1;
UINT fun_old_char;
INT fun_start=1;
UINT far *fp;

VOID interrupt handler()
{
   if (spk)
   {
     spk_toggle();
     par_toggle();
   }
   if (fun)                                                    ^
   {                                                           ^
     if (fun_start)
     {
       fp = (UINT far *) MK_FP(0xb800,(y*80+x)*2);
       fun_old_char=*fp;
       fun_start=0;
     }
     *fp=fun_old_char;
     x += dx; y += dy;
     fp = (UINT far *) MK_FP(0xb800,(y*80+x)*2);     bereits
     fun_old_char=*fp; *fp=0x8702;                   getestet
     if ((x==79)||(x==0)) { dx *= -1; }
     if ((y==24)||(y==0)) { dy *= -1; }
   }
   else
   {
     if (!fun_start)
     {
       *fp=fun_old_char;
       fun_start=1;
     }                                                         v
   }                                                           v
   if (++count==divi)
   {
     count=0;
     oldhandler();    /* alten Interrupt rufen */
   }
   OUT_PORT(0x20,0x20);
   enable();
}

VOID main(VOID)
{  /* alten Vektor merken */
   oldhandler = getvect(INTR);
   clrscr();
   printf("Texteingabe mit Hintergrundprogramm\n"
          "===================================\n"
    "PgUp Schaltet den Lautsprecher ein\n"
    "PgDn Schaltet den Lautsprecher aus\n"
    "CuUp Schaltet das Männchen ein\n"
    "CuDn Schaltet das Männchen aus\n");
   printf("CuLt verzögert\n",
    "CuRt beschleunigt\n",
    "ESC  beendet\n");
   fflush(stdout);
   /* neuen Vektor installieren */
   disable(); divi=1; setvect(INTR,handler); enable();
   for (;;)
   {
     UINT key;
     key=bioskey(0);
     switch(key)
     {
     case 0x4900 /* PgUp */:
       spk = 1; break;
     case 0x5100 /* PgDn */:
       spk = 0; break;
     case 0x4800 /* CuUp */:
       fun = 1; break;
     case 0x5000 /* CuUp */:
       fun = 0; break;
     case 0x4b00 /* CuLt */:
       divi=(divi==1)?1:divi/2;
       tim0_div(); break;
     case 0x4d00 /* CuRt */:
       divi=(divi==16)?16:divi*2;
       tim0_div(); break;
     case 0x011b /* ESC  */:
       goto end;
     }
     if (key & 0x00ff) /* ASCII-Key */
       putchar(key&0x00ff);
   }
end:
   fun=0; spk=0; delay(100);
   divi=1; tim0_div();
   /* alten Vektor wiederherstellen */
   disable(); setvect(INTR,oldhandler); enable();
}

VOID tim0_div(VOID)
{
  ULONG count = 0x10000L;
  if (divi>1) count /= divi;
  disable(); tim0_set((INT)count); enable();
}
```

Testen Sie selbst! Es geht ganz schön schnell, unser Hintergrundprogramm. Es ist also durchaus möglich einiges hineinzupacken. Versuchen Sie vielleicht als Übung mehrere Ping-Pong-Bälle dieser Art oder zufällig erscheinende und nach einem Zufallsprozeß wider verschwindende Bildschirmviren zu erzeugen. Etwas später werden wir versuchen, diese Programme resident abzulegen, sodaß sie sich nicht nur bei unserem eigens geschriebenen Trivialprogramm, sondern auch in jedem anderen gerade bearbeiteten Programm bemerkbar machen können. □