Hardwarenahes Programmieren in C und C++

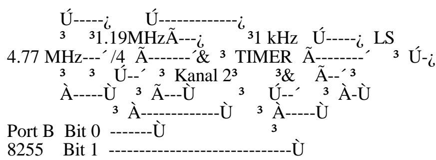
Franz Fiala, N, TGM DSK-401

Teil 6a: Der Lautsprecher im PC

Nachdem wir in den vergangenen Folgen den PC insgesamt besprochen haben, können wir nun ins Detail gehen - und beginnen mit dem Lautsprecher. Ein Bit nur, das wir steuern müssen, und eine Menge Probleme.

Der Lautsprecher ist im PC am gefahrlosesten programmierbar, da seine Einstellung keine weiteren lebenswichtigen Funktionen berührt, wenn man nur ein paar Grundsätze beherzigt. Außerdem wird der Lautsprecher vom BIOS und vom MSDOS nicht weiter unterstützt: wir haben also gleichzeitig die Gelegenheit zu zeigen, wie man eine neue Hardwareeinheit ins System integrieren kann (eine der nächsten Folgen). Bevor wir den Lautsprecher tatsächlich ein- und ausschalten, benötigen wir einige Schaltungsdetails:

Lautsprecherbeschaltung



Der Lautsprecher ist also über Port B des Bausteins 8255 programmierbar. Das BIOS sorgt beim Booten dafür, daß der Port B auch wirklich als Ausgang geschaltet ist. Die Adressen von Port-B entnehmen wir der Tabelle in den **PC-NEWS**-27, S.32, bzw. **PC-NEWS**-30, S.52. Bit 0 schaltet mit 1 den Timer Kanal 2 ein, Bit 1 verbindet den Timer mit dem Lautsprecher. Auch der Timer wurde durch das BIOS bereits voreingestellt, wir müssen uns zunächst um seine Programmierung nicht sorgen.

Timer steuert Lautsprecher

Aus der Skizze sehen wir, daß unabhängig von der tatsächlichen Prozessorgeschwindigkeit Kanal 2 des Timers immer mit einem Viertel von 4,77 MHz getaktet wird. In der Grundeinstellung erzeugt der Kanal 2 des Timers immer das übliche Piepsen mit 1kHz.

Wie erfolgt nun das Einschalten des Lautsprechers in C? Da Port-B die Adresse 0x61 hat, könnte man einfach schreiben:

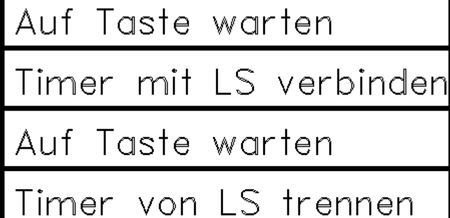
```
outportb(0x61,3);
```

Das wäre zwar - was die erforderlichen Verbindungen in der obigen Skizze betrifft - goldrichtig, aber wir müssen beim Schalten beachten, daß die Bits 2-7 von Port B andere, für den PC lebenswichtige Funktionen haben und daher nicht verändert werden dürfen. Zuerst muß man den Zustand von Port B feststellen und nur die Bits 0 und 1 setzten und dann das Resultat wieder hinausschreiben. Etwa so:

```
c = inportb(0x61); c |=3; outportb(0x61,c);
```

Wie in Folge 4 (**PC-NEWS**-30, S.48) beschrieben wurde, sind die Ein- und Ausgabebefehle oft von Compiler zu Compiler unterschiedlich. Um einigermaßen compilerunabhängig arbeiten zu können, wurden diese Funktionen durch Makros `IN_PORT()` und `OUT_PORT()` ersetzt (erkennbar an der Großschreibweise), die in der Headerdatei `portable.h` ausgeführt sind. Außerdem wurde darauf geachtet, daß alle Typen in Großschrift und teilweise abgekürzt geschrieben werden können. Das wird in der Datei `mytypes.h` sichergestellt. Also kann `VOID` statt `void` und `UCHAR` statt `unsigned char` geschrieben werden. Durch diese strukturierte Schreibweise wird auf die Bedeutung einzelner Bezeichner hingewiesen. Da Typen grundsätzlich an bestimmten Stellen des Programms zu liegen kommen (links von Variablennamen), erhält das Listing eine gut lesbare Struktur. Diese Technik wird vor allem bei Windows-Programmen angewendet.

Mit diesen Vorbemerkungen können wir ein Programm zum Ein- und Ausschalten des Lautsprechers schon schreiben:




```
/* HC06LS0.C */
/*
 * Tonerzeugung durch voreingestellten Timer
 * ========================================
 */
#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <mytypes.h>
#define P_IO
#include <portable.h>

VOID main(VOID)
{
  UCHAR c;

  clrscr();
  printf("Ein- und Ausschalten des Timers "
         "und des Lautsprechers\n\n");
  printf("Taste verbindet den Lautsprecher "
         "dem Timer \n");
  getch();

  c = IN_PORT(0x61);   /* 8255, Port-B */
  c |= 2;              /* LS ein */
  OUT_PORT(0x61,c);

  c = IN_PORT(0x61);   /* 8255, Port-B */
  c |= 1;              /* Timer-Kanal-2 ein */
  OUT_PORT(0x61,c);

  printf("Taste trennt den Lautsprecher "
         "und den Timer\n");
  getch();

  c = IN_PORT(0x61);
  c &= ~1;             /* Timer-Kanal-2 aus */
  OUT_PORT(0x61,c);

  c = IN_PORT(0x61);
  c &= ~2;             /* LS aus */
  OUT_PORT(0x61,c);

  clrscr();
}
```

Die im Programm vorkommenden Sequenzen zum Ein- und Ausschalten des Tones bieten sich für für eine Formulierung in einem Unterprogramm an. Erinnern wir uns, daß wir für die einzelnen Hardwareadressen in der Header-Datei `IODEF.H` eigene Bezeichnungen eingeführt haben, die uns auch bei späterer Lektüre des Programms erklären, was denn eigentlich 0x61 bedeutet. Die gewählte Abkürzung hieß `XT_PPI_B`, was soviel heißt wie: seit dem XT bekannt, PPI ist die Kurzbezeichnung für den 8255 und B ist unser Port.

```
VOID spk_on(VOID)
{
  UCHAR c = IN_PORT(XT_PPI_B); /* Lautsprecher EIN */
  c |= 2;
  OUT_PORT(XT_PPI_B,c);
}

VOID spk_off(VOID)
{
  UCHAR c = IN_PORT(XT_PPI_B); /* Lautsprecher EIN */
  c &= ~2;
  OUT_PORT(XT_PPI_B,c);
}

VOID tim2_on(VOID)
{
  UCHAR c = IN_PORT(XT_PPI_B); /* Timer EIN */
  c |= 1;
  OUT_PORT(XT_PPI_B,c);
}

VOID tim2_off(VOID)
{
  UCHAR c = IN_PORT(XT_PPI_B); /* Timer AUS */
  c &= ~1;
  OUT_PORT(XT_PPI_B,c);
}
```

```
/* HC06LS1.C */
/*
 * Tonerzeugung durch voreingestellten Timer
 * =========================================
 * Mit eigenen Schaltfunktionen
 */
#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <mytypes.h>
#define P_IO
#include <portable.h>
#include <iodef.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\source\spk.c"
#endif

VOID main(VOID)
{
  clrscr();
  printf("Ein- und Ausschalten des Timers "
         "und des Lautsprechers\n\n");
  printf("Taste schaltet den Lautsprecher "
         "und den Timer ein\n");
  getch();

  spk_on(); tim2_on();

  printf("Taste schaltet den Lautsprecher "
         "und den Timer aus\n");
  getch();

  spk_off(); tim2_off();
  clrscr();
}
```

Nachdem wir die nützlichen Unterprogramme aus dem Hauptprogramm herausgelöst haben, fügen wir sie in die Bibliothek `mylib.h` ein. Damit das Hauptprogramm sowohl zum Testen der einzelnen Bibliotheksroutinen als auch zum Testen der fertigen Bibliothek ohne Änderung des Kodes möglich wird, wird ein kleiner Trick angewendet.

Zuerst verlagert man die Routinen `spk_on()`, `spk_off()`, `tim2_on()` und `tim2_off()` in eine Datei `spk.c` im Verzeichnis `SOURCE`. In diese Datei werden auch noch weitere Funktionen kommen. Dann ergänzt man in der Headerdatei `mylib.h` um die beiden neuen Prototypen und merkt an, in welcher Quelldatei sie sich befinden. Um die Bibliothek zu bilden, müßte man jetzt die gesamte Bibliothek neu kompilieren. Das sollte man aber erst tun, wenn man über die Funktionsfähigkeit der neuen Routinen durch eingehende Tests Bescheid weiß. Dieses und die folgenden Programme sind jetzt unsere Testprogramme.

Während dieser Testphase sollen die neuen Funktionen mitkompiliert werden (sie können sich ja bei kleinen Änderungen noch inhaltlich verändern) und später, wenn die Tests abgeschlossen sind, sollen die Testprogramme genauso ausgeführt werden können.

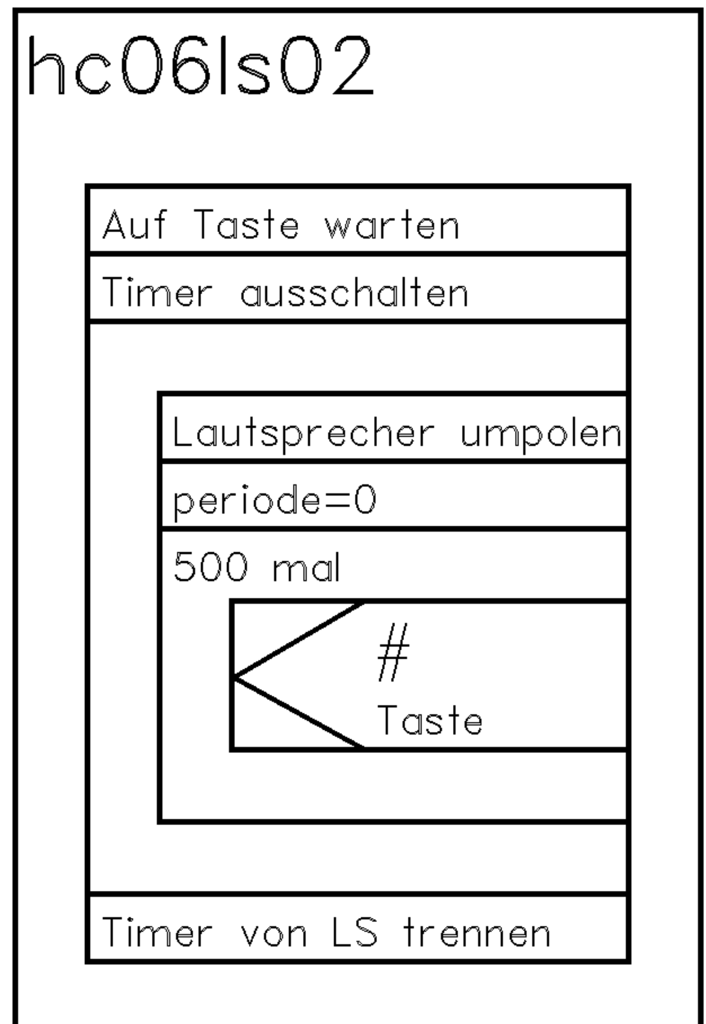
Der kleine Trick besteht darin, daß in der Headerdatei `mylib.h` ein Makro `MYLIB` exisitiert, das - wenn es definiert ist - die Inklusion der Datei `spk.c` verhindert. Während des Testens wird also `spk.c` inkludiert, dann nicht mehr, die Funktionen werden während des Linkens aus der fertigen Bibliothek `MYLIB.LIB` geholt. Die folgenden Zeilen kommen im Programm `HC06LS1.C` dazu; die beiden Funktionen entfallen, siehe das hier nicht abgedruckte Programm `HC06LS1A.C`.

Diesen Vorgang des Auslagerns von Funktionen in eine wachsende Bibliothek wird in der Folge wiederholt in gleicher Weise durchgeführt.

Tonerzeugung durch Softwareverzögerung, HC06LS2.C

Selbstverständlich kann der Ton im Lautsprecher auch durch programmgesteuertes Umpolen am letzten UND-Gatter erzeugt werden. Die Frequenz ergibt sich aus der Verzögerung, die das Programm bewirkt. Es ist das keine empfehlenswerte Methode zur Erzeugung definierter Zeitabstände, denn dazu eignet sich natürlich der Timer wesentlich besser; aber das Ergebnis gibt uns wichtige Aufschlüsse über die Arbeitsweise des Rechners.

Die Verzögerung entsteht durch die `for`-Schleife, die 500 mal durchlaufen wird. Da das bei verschiedenen Rechner verschieden lang dauern wird, muß man hier je nach CPU und Takt eine andere Größe einsetzen, will man dasselbe Ergebnis erhalten. Ein wesentlicher Zeitfaktor der Schleife ist der Aufruf von `bioskey()`, damit man den Ton auch unterbrechen kann.

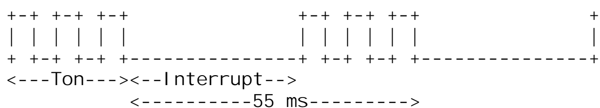


*Anmerkung: Im obigen Struktogramm ist leider ein Fehler passiert, der Abbruch durch Tastendruck muß **nach** der 500mal durchlaufenen Schleife und **nicht innerhalb** erfolgen.*

```
VOID spk_toggle(VOID) /* Lautsprecher umpolen */
{
  UCHAR c = IN_PORT(XT_PPI_B);
  c ^= 2;
  OUT_PORT(XT_PPI_B,c);
}
```

```
/* HC06LS2.C */
/*
 * Ton durch Verzögerungsschleife
 * ==============================
 */
#include <dos.h>
#include <conio.h>
#include <bios.h>
#include <stdio.h>
#include <mytypes.h>
#define P_IO
#include <portable.h>
#include <iodef.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\source\\spk.c"
#endif

VOID main(VOID)
{
  UINT t;             /* Zeitzähler */

  clrscr();
  printf("Erzeugen eines Tones "
         "durch Umpolung des Lautsprechers\n\n");
  printf("Periodendauer durch "
         "Verzögerungsschleife verlängert\n");
  printf("Start mit Taste, Abbruck mit Taste\n\n");
  getch();

  tim2_off();               /* Timer ausschalten */

  for(;;)
  {
    spk_toggle();           /* Lautsprecher umpolen */

    for (t=0; t<500; t++); /* kleine Verzögerung */

    if (bioskey(1))         /* Abbruck mit Taste */
    {
      getch();
      break;
    }
  }

  spk_off();
  tim2_off();               /* LS und Timer trennen */
  printf("Warum 'gurgelt' der Ton?\n");
  getch();
  clrscr();
}
```

Leider ist das Ergebnis nicht ganz 'sauber', der Ton ist keinesfalls rein, es hört sich an, als würde sich auch noch ein anderes Programm bei der Steuerung des Lautsprechers mitmischen. Und genauso ist es! Man ist leider nicht allein auf der Welt, nicht einmal im PC. Interrupts können laufend stattfinden und der Timer-Kanal 1 tut das auch etwa 18mal pro Sekunde.

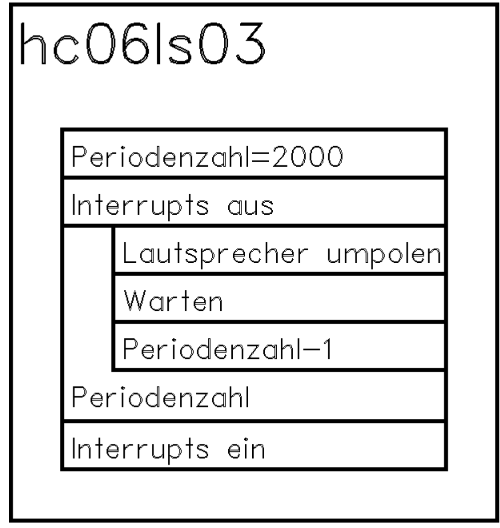


Ausschalten der Interrupts, HC06LS3.C

Was ist zu tun? Glücklicherweise kann man die Interrupts ausschalten. Der C-Compiler stellt uns zwei Funktionen dafür zur Verfügung: `enable()` und `disable()`.

Leider ergibt das eine weitere Schwierigkeit, daß mit dem einfachen Ausschalten aller Interrupts auch derjenige für die Tastatur mitausgeschaltet wird und daher der Abbruch per Tastendruck nicht mehr möglich ist.

Als vorläufige Lösung wird eine Softwarewarteschleife eingesetzt, die den Ton vorzeitig beendet und danach die Interrupts wieder einschaltet. Das Struktogramm ist ein Auszug, der die Handhabung der Interrupts zeigt:




```
/* HC06LS3.C */
/*
 * Ton mit Verzögerungsschleife, ohne Interrupts
 * =============================================
 */
#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <mytypes.h>
#define P_IO
#include <portable.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\SOURCE\\SPK.C"
#endif

VOID main(VOID)
{
  ULONG periodenzahl;
  UINT  t;

  clrscr();
  printf("Erzeugen eines Tones "
         "durch Umpolung des Lautsprechers\n\n");
  printf("Periodendauer durch "
         "Verzögerungsschleife verlängert\n");
  printf("Start mit Taste, "
         "Konstante Anzahl von Perioden\n\n");
  getch();

  spk_on();
  tim2_off();

  periodenzahl = 2000; /* erster Versuch */
  do
  {
    spk_toggle();
    for (t=0; t<1000; t++) ;
    periodenzahl--;
  }
  while (periodenzahl);

  printf("Ton gurgelt noch immer, "
         "Achtung, jetzt Interrupts aus!\n\n");
  getch();

  periodenzahl = 2000;
  disable();
  do
  {
    spk_toggle();
    for (t=0; t<1000; t++) ;
    periodenzahl--;
  }
  while (periodenzahl);
  enable();

  printf("Ton sollte jetzt sauber, "
         "wie beim Timer gewesen sein\n");
  getch();

  /* Lautsprecher und Timer AUS */
  spk_off(); tim2_off();
}
```

Kombiton, HC06LS4.C

Wir haben also zwei unabhängige Möglichkeiten kennengelernt, Töne zu erzeugen: Zuerst die Verbindung des Timers mit dem Lautsprecher, dann die direkte Umpolung des Lautsprechers. Als Gag kann man natürlich beide Möglichkeiten kombinieren, was im Programm `HC06LS4.C` gezeigt wird.

```
/* HC06LS4.C */
/*
 * Kombinierter Ton
 * ================
 */
#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <mytypes.h>
#define P_IO
#include <portable.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\SOURCE\\SPK.C"
#endif

VOID main(VOID)
{
  ULONG zyklen;
  UINT  t;

  clrscr();
  printf("Kombinierter Ton aus Timer+Umpolung\n\n");
  printf("Periodendauer verlängert\n");
  printf("Start mit Taste, "
         "Konstante Anzahl von Perioden\n\n");
  getch();

  printf("Zuerst nur der Timer-Ton (Taste)\n");

  spk_on(); tim2_on();

  printf("Nach Taste kombiniert mit Umpolung, "
         "endet selbst\n");
  getch();
  zyklen = 3000;
  disable();
  do
  {
    spk_toggle();
    for (t=0; t<1000; t++) ;
    zyklen--;
  }
  while (zyklen);
  enable();

  printf("Jetzt wieder nur der Timer (Taste)\n");
  getch();
  /* Lautsprecher und Timer AUS */
  spk_off(); tim2_off();
}
```

6b: Programmierungs des Timers

Eine weit bessere Möglichkeit zur Tonerzeugung stellt uns der Timer zur Verfügung. Wir müssen aber wissen, wie man den Timer programmiert.

Zunächst sollten wir uns darüber klar sein, daß der Timer ein alter Herr ist und man einen solchen wichtigen Baustein heute anders konzipieren würde. Der Timer besteht aus drei voneinender unabhängigen Kanälen, die in unserem PC fest verdrahtet sind. Kanal 0 ist für die Zeitgebung eingestellt löst in Abständen von 53 ms einen Interrupt aus; dieser Kanal wird uns noch später beschäftigen. Kanal 1 ist für den RAM-Refresh verantwortlich und normalerweise nicht verwendbar. Kanal 2 ist auf 1 kHz eingestellt und kann über Gatter mit dem Lautpsrecher verbunden werden; die zugehörige Logik haben wir im ersten Teil kennengelernt.

Alle Kanäle sind über die Adressen 0x40, 0x41 und 0x42 anzusprechen und werden über einen gemeinsamen Steuerkanal 0x43 programmiert. Welcher Timer-Kanal gerade programmiert wird, bestimmen zwei Bits im Steuerwort. Wie die Programmierung erfolgt und was die einzelnen Bits bedeuten zeigt das folgende Programm:

```
/* HC06TI1.C */
/*
 * Timer-Demonstration
 * ===================
 */
#include <stdio.h>
#include <conio.h>
#define P_IO
#include <portable.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\source\spk.c"
#endif

VOID main(VOID)
{
  clrscr();
  printf("Der Timer arbeitet völlig unabhängig "
         "von gleichzeitig laufender Software\n");
  printf("Taste\n");
  getch();
  spk_on(); tim2_on();
  printf("Während der Timer für uns arbeitet, "
         "können wir machen was wir wollen\n"
         "Das ist der normale Timer-Ton mit 1 kHz "
         "(Taste)\n");
  getch();
  printf("Der Timer kann auch umprogrammiert werden!\n"
          "Steuerwort auf 0x43\n"
          "7 6 5 4 3 2 1 0 Bit\n"
          "0 0             Kanal 0 Uhr: 53 ms, "
                                 "0x36, 0x00, 0x00 ??\n"
          "0 1             Kanal 1 DRAM-Refresh, 15 us\n"
          "1 0             Kanal 2 LS: 1 kHz, "
                                 "0xb6, 0x33, 0x05\n"
          "    0 0         Latch\n"
          "    0 1         LSB folgt\n"
          "    1 0         MSB folgt\n"
          "    1 1         LSB-MSB folgt\n"
          "        0 0 0   Interrupt on count\n"
          "        0 0 1   One-Shot\n"
          "        0 1 0   Generator\n"
          "        0 1 1   Rechteck-Generator\n"
          "        1 0 0   Triggered-Strobe-Soft\n"
          "        1 0 1   Triggered-Strobe-Hard\n"
          "              0 Binary\n"
          "              1 BCD\n"
          "Versuchen wir die Zeitkonstante 0x1000 (Taste)\n");
  getch();
  OUT_PORT(0x43,0xb6);
  OUT_PORT(0x42,0x00); /* Zeitkonstante LSB */
  OUT_PORT(0x42,0x10); /* Zeitkonstante MSB */
  printf("Jetzt wieder Originalprogrammierung 0x0533"
         "(Taste)\n");
  getch();

  OUT_PORT(0x43,0xb6);
  OUT_PORT(0x42,0xa8); /* Zeitkonstante LSB */
  OUT_PORT(0x42,0x04); /* Zeitkonstante MSB */

  printf("Taste\n");
  getch();

  spk_off(); tim2_off();
}
```

Die Timer-Kanäle werden beim Einschalten des Rechners durch das BIOS programmiert. Die Zeitkonstante von Kanal 0 wird auf 0 eingestellt. Dabei entsteht die längste Periodendauer von 65536 Taktschritten.

Die Programmierung erfolgt - wie Sie am obigen Beispiel sehen können - so, daß zuerst das Steuerwort in die Adresse 0x43 geschrieben wird und danach die Zeitkonstante mit dem niederweitigen Byte zuerst in das entsprechende Timerregister geschrieben wird. Diese Abfolge ist aber durch Veränderung des Steuerwortes änderbar.

Damit so sensible Vorgänge wie das Steuren des Timers nicht immer wieder neu erfunden werden müssen, werden wir die Kommunikation eigenen Funktion übertragen, die wir in der Datei `tim.c` zusammenfassen:

`tim2_init()` versetzt den Timer-Kanal-2 in den Anfangszustand:

```
VOID tim2_init(VOID)
{
  OUT_PORT(PC_TIM|TIM_CTL,0xb6);
  /* Zeitkonstante niederwertiger Teil */
  OUT_PORT(PC_TIM|TIM_CH2,0x33);
  /* Zeitkonstante höherwertiger Teil */
  OUT_PORT(PC_TIM|TIM_CH2,0x05);
}
```

`tim2_set()` initialisiert den Timer-Kanal-2 auf einen beliebigen Anfangswert; damit können wir die Tonhöhe des ausgegebenen Tones steuern.

```
VOID tim2_set(UINT count)
{
  OUT_PORT(PC_TIM|TIM_CTL,0xb6);
  /* Zeitkonstante niederwertiger Teil */
  OUT_PORT(PC_TIM|TIM_CH2,(UCHAR)count);
  /* Zeitkonstante höherwertiger Teil */
  OUT_PORT(PC_TIM|TIM_CH2,(UCHAR)(count>>8));
}
```

`tim2_read()` liefert den aktuellen Zählerstand des Timers zurück.

```
UINT tim2_read(VOID)
{
  UINT h,l;
  l=IN_PORT(PC_TIM|TIM_CH2);
  h=IN_PORT(PC_TIM|TIM_CH2);
  return ((h<<8)+l);
}
```

`tim2_wait()` wartet `time` Perioden

```
VOID tim2_wait(UINT time)
{
  UCHAR h;

  tim_on();
  tim2_set(0xff00);
  do
  {
    do
    {
      /* niederwerigen Teil lesen und verwerfen */
      h=IN_PORT(PC_TIM|TIM_CH2);
      h=IN_PORT(PC_TIM|TIM_CH2);
    }
    while (h);
  }
  while (time--);
  tim_off(); tim2_init();
}
```

Timer auslesen, HC06TI2.C

Obwohl die Timer vorteilhafterweise interruptgesteuert arbeiten sollten, da der Zählzeitpunkt dann exakt eingehalten wird, kann man auch den Zählerstand der Timer ablesen. Dabei muß man allerdings beachten, daß man nicht einen bestimmten Zählerstand erwarten darf, wie das folgende Programm zeigt.

```
/* HC06TI2.C */
/*
 * Timer auslesen
 * ==============
 */
#include <bios.h>
#include <stdio.h>
#include <conio.h>
#define P_IO
#include <portable.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\source\\spk.c"
#endif

VOID main(VOID)
{
  UINT timer;

  clrscr();
  printf("Der Timer kann auch ausgelesen werden\n");
  printf("Timer auslesen: (Taste)\n");
  getch();
  tim2_on();
  tim2_set(0xffff);
  do
  {
    timer = tim2_read();
    printf("%4x ",timer);
  }
  while (bioskey(1)==0);
  getch();
  tim2_off();
  tim2_init();
  getch();
}
```

Dieses Programm initialisiert den Timer-Kanal-2 auf die längste mögliche Zeitkonstante 0xffff und versucht in einer `do..while`-Schleife den Zählerstand am Bildschirm darzustellen. Wie oft das während eines Zählzyklus gelingt, sehen Sie aus der folgenden Darstellung:

```
a9e2 a146 992a 910c 88e6 80c6 7644 6dfa 65dc 5dc0 55a4 4d88 4566 3d46 3526 2d08
1fb2 171e  e00 f3c8 eb54 e336 da0e d1ce c9aa c18e b96e b152 a930 a112 98f4 90d6
813e 78a0 707e 6862 6042 581e 4ffe 47dc 3fbe 379c 2f7c 275c d4e8 c968 c114 b8f8
aba6 a30c 9aea 92ca 8aa8 8288 7a62 7242 6a26 5b04 538e 4b76 4350 3b36 331a 2afe
1daa 1516  cf6  4d8 fcbc f498 ea16 e1d4 d9ae d190 c96e c152 b934 b10e 9ed6 956e
```

Man sieht daß ein Timer von der Anfangszahl beginnt und nach unten zählt. Erreicht er die Null, löst er je nach Betriebsart ein Signal an seinem Ausgang aus. Im Falle des Kanals 2 polt er dabei den Lautsprecher um. Im Falle des Kanals 0 löst er gemäß Verdrahtung einen Interrupt aus.

Dabei ist aber zu bedenken, welche Konfiguration beim Start des Programms verwendet wird (Hier: 486SL-25, Borland-C in einem Widows-Fenster mit VGA-Display). Andere Konfigurationen können ein ganz anderes Bild ergeben. Im Extremfall (XT, 4MHz, langsames Display) kann man per Programm dem Zählzyklus gar nicht mehr folgen, wenn man zur Darstellung eines bestimmten Zählerstandes länger braucht als der Zähler für einen Durchlauf.

Feststellung des Null-Durchgangs, HC06TI3.C

Wie stellt man auf Grund des Auslesens des Timers fest, wann der Timer 'bei 0 vorbeikommt'? Kanal-2 kann ja wegen der festen Verbindung mit dem Lautsprecher keinen Interrupt auslösen, der uns das präzise melden könnte. Im folgendes Programm wird gezeigt, daß man sich daran orientieren kann, daß der neue Zählerstand grundsätzlich kleiner ist als der alte, da der Timer ein Abwärtszähler ist, nur, wenn die Zählung wieder von vorn beginnt, also der Start-Zählerstand eingelesen wird, ist das nicht der Fall.

```
/* HC06TI3.C */
/*
 * Timer Nulldurchgang feststellen
 * ===============================
 */
#include <bios.h>
#include <stdio.h>
#include <conio.h>
#include <mylib.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\source\\spk.c"
#endif

VOID main(VOID)
{
  UINT timer2;
  UINT timer2alt;

  clrscr();
  printf("Timer nach Nulldurhgang auslesen: (taste)\n");
  getch();
  tim2_on();
  tim2_set(0xffff);

  timer2alt=tim2_read();
  do
  {
    timer2=tim2_read();
    if ((timer2alt>>8)<(timer2>>8))
    {
      printf("%4x ",timer2);
    }
    timer2alt=timer2;
  }
  while (bioskey(1)==0);
  getch();

  tim2_off();
  tim2_init();
  getch();
}
```

Als Ergebnis dieses Programms erhält man (je nach Rechnergeschwindigkeit) folgendes Bild:

```
Timer nach Nulldurhgang auslesen: (taste)
ffda ffda ffce fffc ffd2 fffa  7be ffc2 ff02 ffc4 fff0 ffcc ec58 ffe2 ffe2 ffd4
ff02 fffa ffc4 ffc0 ffd8 ffd6 ffce ffe4 ffec fff8 fff8 ffd8 ff06 ffc2 ffec f356
ffe4 ffca ffd8 ffcc ffd2 ffea ffe8 ffbe ffce ffd6 ffe6 ffce ff06 ffe4 fffe fff6
ffe8 ffe8 ffbe ffdc ffc2 ffce fffa fff6 ff02 ff00 ffe2 ffd2 ffc4 fffc ffe6 fff4
```

Im Prinzip liegen die Erkennungszeiten des Nulldurchgangs bei maximal 256 Taktschritten (0xff02); dabei gibt es aber immer wieder einige Ausreißer (0xec58 und 0x07be), die durch das periodische Auftreten des Interrupts von Kanal 0 hervorgerufen werden. Während dieser Interruptzeiten hat unser Programm eine Zwangspause und kann nicht im normalen Tempo weiterarbeiten. Abhile: Interrupts abdrehen, wenn man die Zeit genau wissen will.

Zeit messen, HC06TI4.C

Wir nutzen also die Fähigkeit des Timers, um kleine Funktionen zu formulieren, die uns eine der Funktion `delay()` ähnliche (aber feiner abgestufte) Zeitverzögerung ermöglicht.

```
VOID tim2_waitus(UINT time)
{
  UINT ende=0xffff-time;

  disable();
  tim2_on();
  tim2_set(0);

  do
  {
  }
  while (tim2_read()>ende);

  tim2_off(); tim2_init();
  enable();
}

VOID tim2_waitms(UINT time)
{
  disable();

  do
  {
    tim2_waitus(1193);
  }
  while (time--);

  enable();
}

VOID tim2_waits(UINT time)
{
  disable();

  do
  {
    tim2_waitms(1000);
  }
  while (time--);

  enable();
}
```

Beachten Sie, daß jeweils beim Eintreten in die jeweiligen Funktionen die Interrupts mit `disable()` ausgeschaltet werden und beim Verlassen mit `enable()` wieder eingeschaltet werden.

Die Grundverzögerung wird in der innersten Routine `tim2_waitus()` erreicht. Die Zähleinheit ist nicht genau eine Mikrosekunde sondern 0,84 us=1/1200000 s. Daher muß der Multiplikator der Funktion `tim2_waitms()` auch 1193 betragen.

Diese Routinen sollten nur sparsam verwendet werden, halten sie doch die interne Uhrzeit wegen der Abschaltung der Interrupts an. Allzugroße Genauigkeit im Mikrosekundenbereich darf auch nicht erwartet werden.

```
/* HC06TI4.C */
/*
 * Zeitverzögerung
 * ===============
 */
#include <stdio.h>
#include <conio.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\source\\spk.c"
#endif

VOID main(VOID)
{
  UINT count = 0;

  clrscr();
  printf ("Warten mit tim2_waitus (Taste)\n");
  getch();

  do
  {
    tim2_waitus(1000);
    count++;
  }
  while (count<5000);

  printf ("5 Sekunden gewartet\n\n");

  printf ("Warten mit tim2_waitms() (Taste)\n");
  getch();

  tim2_waitms(10000);

  printf ("10 Sekunden gewartet\n\n");

  printf ("Warten mit tim2_waits() (Taste)\n");
  getch();

  tim2_waits(10);

  printf ("10 Sekunden gewartet\n");
  tim2_init();
  getch();
}
```

Wir kennen den Timer jetzt ausreichend gut, um ihn erfolgreich zur gezielten Steuerung des Lautsprechers verwenden zu können.

6b: Töne macht der Timer

Programmiert man den Timer wie angegeben und verbindet man gleichzeitig den Timer mit dem Lautsprecher (`spk_on()`) und schaltet den Timer ein (`tim2_on()`), dann übernimmt der Timer die Tonerzeugung und das laufende Programm muß sich nicht mehr darum kümmern.

Gleitender Ton, HC06LS5.C

Das erste Beispiel ist ein Programm zur Erzeugung eines in der Frequenz ansteigenden Tones.

```
/* HC06LS5.C */
/*
 * Gleitende Töne
 * ==============
 */
#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\source\\spk.c"
#endif

VOID main(VOID)
{
  UINT count_start = 0x0100;
  UINT count_stop = 0x1000;
  UINT count = count_start;

  clrscr();
  spk_on();
  tim2_on();
  tim2_init();
  printf("Gleitende Töne, pro Ton 50ms \n");
  printf("Zunächst der normale Timer-Ton mit 1 kHz. "
         "Gleitton beginnt mit Taste\n");
  getch();

  do
  {
    tim2_set(count);
    delay(50);
    count +=20;
  }
  while(count<count_stop);

  printf("\nNormaler Timer-Ton mit 1 kHz (Taste)\n");
  getch();
  tim2_init();
  printf("Ende mit (Taste)\n");
  getch();
  tim2_off();
  spk_off();
  tim2_init();
}
```

Nach 50 ms wird die Periodendauer des Tones um 20 erhöht.

Rauschen, HC06LS6.C

Eine zufällig gewählte Periodendauer wird 1 ms lang ausgegeben.

```
/* HC06LS6.C */
/*
 * Rauschen
 * ========
 */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\source\\spk.c"
#endif


VOID main(VOID)
{
  UINT rand_start = 0x0050;
  UINT rand_stop = 0x1000;
  UINT count;

  clrscr();
  printf("Rauschen\n");
  printf("Taste\n");
  getch();
  spk_on(); tim2_on();
  do
  {
    count = random(rand_stop-rand_start) + rand_start;
    tim2_set(count);
    delay(1);
  }
  while(!kbhit());

  printf("\nNormaler Timer-Ton mit 1 kHz (Taste)\n");
  getch();
  tim2_init();

  printf("Ende mit Taste\n");
  getch();
  tim2_off();
  spk_off();
  tim2_init();
}
```

Töne als Frequenz angeben, HC06LS7.C

Der Timer wird in Vielfachen der Timer-Taktfrequenz programmiert. Für viele Anwednung ist aber das Programmieren mit einer Frequenz naheliegender. In Anlehnung an die `sound()`-Funktion der BORLAND-Bibliothek wird eine Funktione `soundf()` formuliert, die ebenfalls eine Frequenz als Argument annimmt aber - ein Unterschied muß sein - mit einer Gleitkommazahl als Argument. Als Unterscheidung dient das nachgestellte `f`. Beachten Sie, daß in einer reinen C++-Bibliothek kein Unterschied im Namen bestehen müßte, da der unterschiedliche Typ im Argument zur Unterscheidung ausreicht. Die Ausschaltfunktion `nosoundf()` enthält neben den eigentlichen Schaltern für Timer und Lautsprecher auch noch die Initialisierung des Timers auf den Anfangswert mit `tim2_init()`, damit sich eine programmgesteuerte Veränderung nicht auf Systemebene auswirkt.

```
VOID soundf(FLOAT frequenz)
{
  UINT periode = (UINT)(1192500.0/frequenz);
  tim2_set(periode);
  spk_on();
  tim2_on();
}

VOID nosoundf(VOID)
{
  spk_off();
  tim2_off();
  tim2_init();
}
```

```
/* HC06LS7.C */
/*
 * Frequenzeingabe
 * ===============
 */
#include <dos.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\source\\spk.c"
#endif

VOID main(VOID)
{
  soundf(1000.0);
  delay(500);
  soundf(2000.0);
  delay(500);
  soundf(3000.0);
  delay(500);
  soundf(4000.0);
  delay(500);
  nosoundf();
}
```

Frequenz und Dauer kombiniert, HC06LS8.C

Verwendet man Töne häufiger, ist das abwechselnde Aufrufen von `soundf()` und `delay()` lästig, `sounf1()` kombiniert beides und beendet den Ton auch regulär.

```
VOID soundf1(FLOAT frequenz, UINT dauer)
{
  soundf(frequenz);
  delay(dauer);
  nosoundf();
}
```

```
/* HC06LS8.C */
/*
 * Variable Dauer
 * ==============
 */
#include <mylib.h>
#ifndef MYLIB
#include "..\\source\\spk.c"
#endif

VOID main(VOID)
{
  soundf1(1000.0,500);
  soundf1(2000.0,500);
  soundf1(3000.0,500);
  soundf1(4000.0,500);
}
```

PC als Instrument, HC06LS9.C

Um jetzt noch aufeinander abgestimmte Töne unserer üblichen Instrumentenstimmung verwenden zu können, genügt eine Frequenztabelle. (Natürlich könnte man jede Frequenz dieser Tabelle bei Bedarf jeweils neu berechnen, eine Tabelle spart aber Rechenzeit.)

```
FLOAT freq_tab[] =
{
  130.8,  /*        C          0  */
  138.6,  /*   Db        C#    1  */
  146.8,  /*        D          2  */
  155.6,  /*   Eb        D#    3  */
  164.8,  /*        E          4  */
  174.6,  /*        F          5  */
  185.0,  /*   Gb        F#    6  */
  196.0,  /*        G          7  */
  207.7,  /*   Ab        G#    8  */
  220.0,  /*        A          9  */
  233.1,  /*   Hb        A#    10 */
  246.9,  /*        H          11 */
  261.7,  /*        c          0 */
  277.2,  /*   db        c#    */
  293.7,  /*        d          */
  311.1,  /*   eb        d#    */
  329.6,  /*        e          */
  349.2,  /*        f          */
  370.0,  /*   gb        f#    */
  392.0,  /*        g          */
  415.3,  /*   ab        g#    */
  440.0,  /*        a          */
  466.2,  /*   hb        a#    */
  493.9,  /*        h          */
  523.3,  /*        c1         */
  0.0,
  0.0
};
```

Das folgende Programm zeigt wie man diese Tabelle dazu benutzt, eine chromatische, eine Dur- und eine Moll-Tonleiter zu spielen.

```
/* HC06LS9.C */
/*
 * Verschiedene Tonleitern im PC
 * ============================
 */
#include <stdio.h>
#include <conio.h>
#include <mylib.h>
#ifndef MYLIB
#include "..\\source\\spk.c"
#endif

VOID main(VOID)
{
  FLOAT *frequenz;
  INT tonnummer;
  UINT tondauer=200;

  clrscr();

  printf("Chromatische Tonleiter C1..C..c, "
         "Start mit Taste\n\n");
  getch();
  frequenz = &freq_tab[0];
  do
  {
    soundf1(*frequenz,tondauer);
    frequenz++;
  }
  while (*frequenz>1.0);
  printf("\n\n");

  printf("\nDUR-Tonleiter C1..C..c, "
         "Start mit Taste\n\n");
  getch();
  frequenz = &freq_tab[0];
  tonnummer = 0;
  do
  {
    soundf1(*frequenz,tondauer);
    switch (tonnummer)
    {
    case 0: case 2: case 5: case 7: case 9:
      frequenz++; tonnummer++;
    default:
      frequenz++; tonnummer++;
    }
    tonnummer %= 12;
  }
  while (*frequenz>1.0);
  printf("\n\n");

  printf("\nMOLL-Tonleiter C1..C..c, "
         "Start mit Taste\n\n");
  getch();
  frequenz = &freq_tab[0];
  tonnummer = 0;
  do
  {
    soundf1(*frequenz,tondauer);

    switch (tonnummer)
    {
    case 0: case 3: case 5: case 8: case 10:
      frequenz++; tonnummer++;
    default:
      frequenz++; tonnummer++;
    }
    tonnummer %= 12;
  }
  while (*frequenz>1.0);
  printf("\n\n");

  getch();
}
```

Das wärs zunächst über den Lautsprecher; wir werden ihn aber noch mehrmals antreffen, etwa beim Schreiben einer BIOS oder DOS-Erweiterung. Auch für den Timer werden wir eine eigene C++-Klasse entwerfen, die genau weiß, welche Einstellung der Timer hat, denn derzeit kann man die Register des Timers nicht zurücklesen.

In der nächsten Folge werden Hardware-Interrupts und einfache Interrupt-Service-Routinen geschrieben. ☐

Real programmers don't use LISP. Only effeminate programmers use more parentheses than actual code.

Real programmers disdain structured programming. Structured programming is for compulsive, prematurely toilet-trained neurotics who wear neckties and carefully line up sharpened pencils on an otherwise uncluttered desk.

Real programmers have no use for managers. Managers are a necessary evil. Managers are for dealing with personnel bozos, bean counters, senior planners and other mental defectives.

Real programmers scorn floating point arithmetic. The decimal point was invented for pansy bedwetters who are unable to "think big."

Real programmers like vending machine popcorn. Coders pop it in the microwave oven. **Real programmers** use the heat given off by the cpu. They can tell what job is running just by listening to the rate of popping.

Real programmers don't drive clapped-out Mavericks. They prefer BMWs, Lincolns or pick-up trucks with floor shifts. Fast motorcycles are highly regarded.

Real programmers don't believe in schedules. Planners make up schedules. Managers "firm up" schedules. Frightened coders strive to meet schedules. **Real programmers** ignore schedules.

Real programmers don't like the team programming concept. Unless, of course, they are the Chief Programmer.

Real programmers know every nuance of every instruction and use them all in every real program. Puppy architects won't allow execute instructions to address another execute as the target instruction. **Real programmers** despise such petty restrictions.

Real programmers don't bring brown bag lunches to work. If the vending machine sells it, they eat it. If the vending machine doesn't sell it, they don't eat it. Vending machines don't sell quiche.