Hardwarenahe Programmierung

Teil 5: Hauptspeicherzugriffe

Franz Fiala, N, TGM DSK-372\HARD5.LZH

In dieser Reihe bereits erschienen:

PC·NEWS-26: Teil 1 Eine Rundschau

PC·NEWS-27: Teil 2 Der PC

PC·NEWS-28: Teil 3 Sprache für hardwarenahe Programmierung

PC·NEWS-30: Teil 4 Hardwareprogrammierung, IO-Zugriffe

Genauso, wie die Ein/Ausgabe in die Register der Peripheriebausteine, können wir den Speicher des PC direkt lesen und schreiben. Überlegen wir, wann wir wohin ungestraft zugreifen können (Lesen wird ja wohl überall erlaubt sein! oder?). Die Aufteilung des Hauptspeichers kennen wir schon aus unserem zweiten Teil (PC·NEWS-92/2, S.33). Unser Programm steht irgendwo in der TPA (Transient Program Area) aber wo?

Segmentadresse, HC05MM1.CPP (und HC05MM1.C)

Innerhalb eines C-Programms erfolgt der Zugriff auf Speicheradressen über dem Umweg über Variablen, deren Adresse man eigentlich gar nicht kennt. Wie kann man nun die **Adresse des eigenen Programms** feststellen? Dazu exisitieren in jeder Sprache eigene, i.a. maschinenabhängige Verfahren. C bietet die Funktion `segread()` zur Feststellung der Segmentadressen an.

Das folgende Programm zeigt den Wert der Segmentregister an. **Achtung**: Wenn man das Speichermodell unter '`Optionen`' verändert, wird das Programm nicht noch einmal kompiliert, da aus der Sicht der Kompilers das vorher kompilierte Programm durchaus noch gültig ist, daher nach dem Ändern des Modells die `.EXE`-Datei löschen oder mit `Build-All` neu compilieren.

```
/* HC05MM1.CPP */
/*
 * Wo bin ich im Hauptspeicher ?
 * =============================
 */
#include <dos.h>
#define P_STREAMS
#include <portable.h>
#include <mytypes.h>

VOID main(VOID)
{
  SREGS s;
  constream con;
  con.clrscr();
  con << "Wo bin ich, im Hauptspeicher?\n"
         "Versuchen Sie dieses Programm mit "
         "allen Speicher-Modellen auszuführen!\n";
  segread(&s);
  con << "CS: " << hex << s.cs << endl;
  con << "DS: " << hex << s.ds << endl;
  con << "ES: " << hex << s.es << endl;
  con << "SS: " << hex << s.ss << endl;
  cin.get();
  con.clrscr();
}
```

Adresse einer Variablen

Die Adresse jeder C-Variablen kann über den Adreßoperator & bestimmt werden.

```
  int i;
...
  printf("Adresse von i ist %p\n", &i);
```

Je nach gewähltem Speichermodell ist diese Adresse ein near- oder far-Zeiger. Nur in den großen Speichermodellen (MEDIUM, LARGE, HUGE) kennt man mit dem Adreßoperator auch gleichzeitig die physikalische Adresse, da im Pointerwert eines `far`-Pointers sowohl das Segment als auch der Offset der Adresse enthalten sind. Bei den kleinen Kode-Speicher-Modellen (TINY, SMALL, COMPACT) ist aber diese Adresse nur der Offset, das zugehörige Datensegment muß mit `segread()`, wie im vorigen Beispiel gebildet werden.

Wie ist der Compiler eingestellt?

Um feststellen zu können, in welcher Betriebsart ein Kompiler arbeitet, verwendet man vordefinierte Makros, die entweder das Speichermodell (`__TINY__`, `__SMALL__`, `__COMPACT__`, `__MEDIUM__`, `__LARGE__`, `__HUGE__`) oder die Pointerlänge (`__SDATA__`, `__SCODE__`, `__LDATA__`, `__LCODE__`) spezifizieren. Während die Makros für das Speichermodell immer verfügbar sind, kennen manche Kompiler die Makros für die Pointerlänge nicht. In der Headerdatei `portable.h` sind sie folgendermaßen definiert:

```
/*-------------------------------------------------
    Symbols for data-pointer-type
    __SDATA__  __LDATA__
    Symbols for code-pointer-type
    __SCODE__  __LCODE__
-------------------------------------------------*/
#ifndef __DOS_H
    #include <dos.h>
#endif

#if !defined(__SDATA__) && !defined(__LDATA__)
  #if defined(__MEDIUM__) || \
      defined (__SMALL__) || \
      defined (__TINY__)

    #define __SDATA__ 1

  #endif
  #if defined(__COMPACT__) || \
      defined (__LARGE__) || \
      defined (__HUGE__)

    #define __LDATA__ 1

  #endif
#endif
#if !defined(__SCODE__) && \
    !defined(__LCODE__)

  #if defined(__COMPACT__) || \
      defined (__SMALL__) || \
      defined (__TINY__)

    #define __SCODE__ 1

  #endif
  #if defined(__MEDIUM__) || \
      defined (__LARGE__) || \
      defined (__HUGE__)

    #define __LCODE__ 1

  #endif
#endif
```

Modellunabhängige Adreßbestimmung

Ein Kode, der die physikalische Adresse unabhängig vom gewählten Speichermodell immer richtig bestimmt, muß daher in Abhängigkeit vom eingestellten Speichermodell kompiliert werden. Eine solche Technik benötigt man, wenn man Bibliotheken aufbaut, die für jeden späteren Betriebsfall geeignet sind.

```
/* HC05AD1.CPP */
/*
 * Modellunabhängige Bestimmung der physikalischen Adresse
 * einer Variablen
 */
#include <stdio.h>
#include <conio.h>

#define P_POINTER
#include <portable.h>

void main(void)
{
  int variable;

#ifdef __SDATA__
  struct SREGS s;
#endif

  printf("Die Adresse der Variablen variable ist ");

#ifdef __SDATA__
  segread(&s);
  printf("%4x:%4x\n",s.ds,&variable);
#endif

#ifdef __LDATA__
  printf("%lp\n",&variable);
#endif
  getch();
}
```

Der genaue Wert einer Variablenadresse ist - wenn es sich um in C definierte Variablen handelt - meistens ohne Bedeutung. Anders ist es, wenn man auf genau definierten Adressen im Speicher, etwa im BIOS-Datenbereich oder im Bereich der Interrupt-Vektoren lesen oder schreiben will.

Zugriff auf absolute Speicheradressen

Das in C verwendete Zugriffsverfahren auf absolute Speicheradressen basiert auf der Anwendung von far-Zeigern mit 20-bit-Adresse, bestehend aus Segment und Offset, deren Ziel mit dem dafür vorgesehenen Makro `MK_FP()` eingestellt wird. Umgekehrt können Segment und Offset eines far-Zeigers mit den Makros `FP_OFF()` und `FP_SEG()` wieder zerlegt werden. Das folgende Makro aus `portable.h` definiert `MK_FP()`, wenn es nicht exisitert:

```
#if !defined(MK_FP)
    #define MK_FP(seg,off) \
      ((void far *)(((long)(seg)<<16)|(UINT)(off)))
#endif
```

Die Speicherbereiche im PC werden von verschiedenen Programmteilen kontrolliert. Teilweise ist ihre Position CPU-bedingt (Interrupt-Vektoren 00000..003ff oder Reset-Vektor ffff0), teilweise BIOS-bedingt (BIOS-Daten 00400..004ff), teilweise Hardware-bedingt (diverse BIOS, Video-Bereich a0000..bffff), teilweise durch DOS (TPA xxxxx..9ffff), teilweise durch neuere Programmkonzepte (HIMEM 100000..10ffff, Upper-Memory-Blocks z.B. EMM386 a0000..fffff).

DOS stellt dem aktuellen Programm einen Speicherblock zur Verfügung und steuert diesen Speicherblock durch einen Memory-Control-Block (MCB). Über diese MCBs werden wir später mehr erfahren. Wenn also das eigene Programm gestartet wird, könnte es interessant sein, zu wissen, wo es sich befindet. Wann das wichtig ist? Zum Beispiel, wenn man einer BIOS-Routine Parameter zu übergeben hat, die sich aus Adressen von Variablen oder Programmen eines C-Programmes zusammensetzen.

Speicherblöcke beginnen immer auf ganzzahligen Vielfachen von 16. Daher benötigt man nur eine Segmentadresse, der Offsetanteil des Programmbeginns ist immer 0.

Das Lesen und Schreiben im Hauptspeicher erfolgt immer über einen Pointer, der mit `MK_FP()` gebildet wird.

Betrachten wir beispielsweise den Bildschirm! Ab Adresse 0xb8000 beginnt der Bildschirmspeicher, mit jeweils einem Byte Zeichenstelle und einem Byte Attribut. Der Typ des zu definierenden Pointers wird durch die Daten bestimmt, die man zuzugreifen gedenkt. Je nach Typ des Pointers ist die Wirkung der Pointerarithmetik unterschiedlich. In unserem Fall bieten sich drei Möglichkeiten an:

Man betrachtet den Inhalt des Bildschirmspeichers als eine Folge

a. ganzer Zahlen ohne Vorzeichen

Man erhält mit jedem Zugriff eine ganze Zahl, deren niederwertiger Teil das Zeichen und deren höherwertiger Teil das Attribut ist.

b. von Bytes

Man erhält bei einem Zugriff entweder das Zeichen oder das Attribut

c. von Strukturvariablen

Man erhält beides: Zeichen und Attribut und muß sich nicht mehr darum kümmern, wie man die Anteile zu trennen hat wie bei a.

```
/* HC05MM2A.CPP */
/*
 * Bildspeicher x 3
 * ================
 */
#include <dos.h>
#define P_STREAMS
#include <portable.h>
#include <mytypes.h>

typedef struct
{
  UCHAR Text;
  UCHAR Attr;
} BILDP;

VOID main(VOID)
{
  UINT far * uifp;   // Pointer auf ganze Zahlen
  UCHAR far * ucfp;  // Pointer auf Bytes
  BILDP far * bfp;   // Pointer auf Strukturen aus Text
  UCHAR c,a;         //   und Attribut

  constream con;
  con.clrscr();
  con << "A Zeichen links oben am Bildschirm: " << endl;
  uifp = (UINT far *)MK_FP(0xb800,0x0000);
  ucfp = (UCHAR far *)MK_FP(0xb800,0x0000);
  bfp = (BILDP far *)MK_FP(0xb800,0x0000);

  // a.) Bild besteht aus einer Folge ganzer Zahlen
  c = (UCHAR)(*uifp);
  a = (UCHAR)((*uifp)>>8);
  con << "  Zeichen : " << hex << setw(2) << (UINT)c
      << "  Attribut: " << hex << setw(2) << (UINT)a << endl;

  // b.) Bild besteht aus einer Folge von Bytes
  c = *ucfp;
  ucfp++;
  a = *ucfp;
  con << "  Zeichen : " << hex << setw(2) << (UINT)c
      << "  Attribut: " << hex << setw(2) << (UINT)a << endl;

  // c.) Bild besteht aus einer Folge von Strukturvariablen
  c = bfp->Text;
  a = bfp->Attr;
  con << "  Zeichen : " << hex << setw(2) << (UINT)c
      << "  Attribut: " << hex << setw(2) << (UINT)a << endl;

  con << "Ende mit Taste";
  cin.get();
}
```

Es ist nicht zu übersehen, daß es am übersichtlichsten ist, Strukturen für häufig vorkommende Variablenanordnungen zu definieren, statt einen Bereich geschlossen anzusprechen, wie unter a. oder sich auf eine byteweise Analyse, wie unter b. einzulassen.

In ähnlicher Weise verfährt man bei allen festen Speicheradressen.

Absolut adressierbare Speicherbereiche

Bevor wir das tun, sollten wir uns über die Speicherbereiche informieren, die es gibt: **PC·NEWS**-92/2, S.33-34. Jetzt ist es an der Zeit diese auch detailierter zu behandeln.

Interruptvektoren

Zunächst sind da die 256 **Interruptvektoren** ab Adresse 0, jeweils 4 Bytes pro Vektor, daher 1 kByte insgesamt. Jeder Vektor besteht aus 2 Bytes Offset und Segment, hier wieder immer niederwertiges Byte zuerst. Diese 256 Vektoren werden in einem anderen Teil unserer Folge genauer behandelt.

Was sind Interruptvektoren?

Vektoren sind eine bewährte Möglichkeit, Tabellen und Programme auf verschieblichen Adressen unterzubringen und dennoch durch den Vektor eine ortsfeste Referenz darauf zu haben. Die ursprüngliche Aufgabe von Interruptvektoren ist, daß sie durch Hardwareereignisse ausgelöst werden und unabhängig vom gerade laufenden Programm ein kurzes Interruptserviceprogramm ausführen, welches das laufende Programm unmerklich unterbricht und danach wieder zu diesem zurückkehrt. D.h. der Benutzer bemerkt (fast) nichts von diesen Hintergrund-'Heinzelmännchen', es sei denn, es gibt zu viele dieser Unterbrechungen, dann kann die Vordergrundaktivität auch einmal stark verlangsamt werden.

Die Interruptvektoren bei 80x86 CPUs beginnen bei der absoluten Adresse 0x00000 und enden bei der absoluten Adresse 0x003ff. Diese Anordnung ist durch den Befehlssatz der CPU vorgegeben und kann nicht geändert werden. Jeder Vektor besteht aus 4 Bytes, es gibt daher 256 oder 0x100 Vektoren. Ein Interruptvektor besteht aus Segment und Offset in umgekehrter Anordnung und auch die Halbbytes von Segment und Offset sind jeweils umgekehrt, das niederwertige Byte kommt zuerst.

```
+=======+
|off lo| <---Adresse des Vektors = 4 * Interruptnummer = a
+-------+
|off hi| <--- a+1
+=======+
|seg lo| <--- a+2
+-------+
|seg hi| <--- a+3
+=======+
```

Jeder Vektor belegt 4 Bytes im Speicher. Normalerweise enthält ein Interruptvektor eine Adresse eines Interruptserviceprogramms.

Befehl zum Aufruf der Interruptserviceroutine in Assembler

```
  INT Interruptnummer
```

oder in C

```
  int86(int intno,...);
```

Die Aufgabe, die diesen Interrupts zukommt, ist bei den ersten bereits durch die CPU-Konzeption vorgegeben, alle weiteren zunächst durch das BIOS dann durch das jeweils geladene Betriebssystem, oder zusätzlich geladene Treiber (Maus, Netzwerk, EMS, XMS..). Jedes geladene Programm kann die Interrupts so beibehalten wie sie sind oder diese ändern und auf Routinen umlenken, die erst später geladen werden. Die frisch geladenen Routinen können vor/nach/statt dem alten Treiber ausgeführt werden.

Einige dieser Vektoren sind nicht Zeiger auf Programme sondern auf Daten. Diese wollen wir hier besprechen.

Interrupt Nummer	Adresse	Inhalt
0x1D	0x00074	Video Initialisierung
0x1E	0x00078	Diskette Parameter Table
0x1F	0x0007C	CGA Graphics Font
0x41	0x00104	Disk #1 Parm Table
0x43	0x0010C	EGA Parm Table Ptr
0x44	0x00110	EGA Graphics Font
0x46	0x00118	Disk #2 Parm Table

Diese Adressen wurden dem XT-AT-Handbuch (siehe **PC-NEWS**-31, Seite 10) entnommen.

Unteruchen von Speicheradressen mit DEBUG

Wie kann man nun diese Daten ansehen? Händisch ganz einfach: Mit DEBUG! Beispiel Disketten-Parameter-Tafel 2, die die Parameter der Festplatten enthält:

```
C:\>DEBUG
-D0:118
0000:0110                         01 E5 00 F0 E7 9B 00 F0           ........
0000:0120  E7 9B 00 F0 E7 9B 00 F0-E7 9B 00 F0 E7 9B 00 F0   ................
0000:0130  E7 9B 00 F0 E7 9B 00 F0-E7 9B 00 F0 E7 9B 00 F0   ................
0000:0140  E7 9B 00 F0 E7 9B 00 F0-E7 9B 00 F0 E7 9B 00 F0   ................
0000:0150  E7 9B 00 F0 E7 9B 00 F0-E7 9B 00 F0 E7 9B 00 F0   ................
0000:0160  E7 9B 00 F0 E7 9B 00 F0-E7 9B 00 F0 E7 9B 00 F0   ................
0000:0170  E7 9B 00 F0 E7 9B 00 F0-E7 9B 00 F0 E7 9B 00 F0   ................
-df000:e500
F000:E500  00 D1 03 05 00 00 2C 01-00 00 00 00 00 D1 03 11   ......,.........
F000:E510  00 D1 03 07 00 00 FF FF-00 00 00 00 00 D1 03 11   ................
F000:E520  00 00 04 07 00 00 00 02-00 00 00 00 00 FF 03 11   ................
F000:E530  00 DD 02 05 00 00 2C 01-00 00 00 00 00 DC 02 11   ......,.........
F000:E540  00 DD 02 07 00 00 2C 01-00 00 00 00 00 DC 02 11   ......,.........
F000:E550  00 DD 02 05 00 00 2C 01-00 00 00 00 00 DD 02 11   ......,.........
F000:E560  00 32 01 04 00 00 00 00-00 00 00 00 00 50 01 11   .2..........P..
F000:E570  00 64 02 04 00 00 31 01-00 00 00 00 00 97 02 11   .d....1.........
-q
```

Die ab F000:E500 stehenden Daten sind die Kennwerte der Festplattenparameter. Typisch ist die letzte Eintragung 0x11 = 17 = Anzahl der Sektoren.

Liste der Interruptvektoren, HC05MM4.CPP, MM4.C, MM4A.C

Da die Anzeige aller Interruptvektoren gleichzeitig auf einem Bildschirm nicht möglich ist, werden hier zwei unterschiedliche Lösungen gezeigt. In HC05MM4.CPP erfolgt die Anzeige der 256 Vektoren in 4 Bildschirmseiten, nach jedem Bildschirm wird auf eine Taste gewartet. HC05MM4A.C erlaubt die Eingabe einer Vektornummer (dezimal), ab der die nächsten 64 Vektoren angezeigt werden.

```
/* HC05MM4.CPP */
/*
 * Anzeige der Interruptvektoren
 * =============================
 */

#include <mytypes.h>
#include <constrea.h>
#define P_POINTER
#include <portable.h>

#define COLW 13
#define COL 4
#define ROW 16

VOID main(VOID)
{
  INT num, row, col;
  ULONG far * ip;

  constream con;
  con.clrscr();
  con << "Interruptvektoren\n";

  ip=(ULONG far *)MK_FP(0,0);
  for (num=0; num<0x100; num++)
  {
    col=(num/ROW)%COL; row=num%ROW;
    con << setxy(1+col*COLW,row+2) << hex << setw(2)
        << num << ':'
        << setw(8) <<setfill('0') << *ip;
    ip++;
    if ((col==(COL-1)) && (row==(ROW-1)))
    {
      con << setxy(1,wherey()+1) << "Taste";
      cin.get();
    }
  }
}
```

und wie geht's in C?

```
/* HC05MM4.C */
/*
 * Anzeige der Interruptvektoren
 * =============================
 */
```

```
...
  for (num=0; num<0x100; num++)
  {
    col=(num/ROW)%COL; row=num%ROW;
    gotoxy(1+col*COLW,row+2);
    cprintf("%2x %Fp",num,*ip);
    ip++;
    if ((col==(COL-1)) && (row==(ROW-1)))
    {
      gotoxy(1,wherey()+1);
      cprintf("Taste");
      getch();
    }
  }
...
```

```
/* HC05MM4A.C */
/*
 * Liste der Interruptvektoren
 * ===========================
 */
```

```
#include <stdlib.h>
#include <stdio.h>

void main (int argc,char *argv[])
{
  int far *p;
  int i,i0,off,seg;

  if (argc==1)
    i0=0;
  else
    i0=atoi(argv[1]);

  printf ("\n\n\nListe der Interrupt-Vektoren\n");
  printf ("===========================\n");
  printf ("Aufruf: intvek      "
          "Die ersten 64 Vektoren \n" \
          "        intvek iii  "
          "Die Vektoren beginnend bei iii\n\n");
  for (i=i0;i<i0+64;i++)
  {
    p=(int far *)(long)(i*4);
    off=*p;p++;seg=*p;p++;
    printf ("%2x %3i %4x:%4x",i,i,seg,off);
    if ((i+1)%4) printf ("  "); else printf ("\n");
  }
  printf ("\n");
}
```

BIOS-Datensegment

Nach den Interruptvektoren an der Adresse 00400=40:0=0:400 beginnt der 256 Byte große BIOS-Datenbereich, der in seiner Position niemals verändert wird, daher auf absoluten Adressen immer ansprechbar ist (anders als Variablen eines C-Programms, die ja immer andere Adressen aufweisen). Das Feststehen der Adressen hat auch unmittelbar zur Folge, daß die BIOS-Funktionen nicht 'reentrant' sind, sich also nicht selbst aufrufen können und auch nicht von verschiedenen Programmen gleichzeitig benutzbar sind (Multitasking).

BIOS-Datensegment

```
40:00  word    COM1 port address |   These addresses are zeroed out in the
40:02  word    COM2 port address |   OS/2 DOS Compatibility Box if any of
40:04  word    COM3 port address |   the OS/2 COM??.SYS drivers are loaded.
40:06  word    COM4 port address |
40:08  word    LPT1 port address
40:0A  word    LPT2 port address
40:0C  word    LPT3 port address
40:0E  word    LPT4 port address        (not valid in PS/2 machines)
40:0E  word    PS/2 pointer to 1k extended BIOS Data Area at top of RAM
40:10  word    equipment flag (see int 11h)
               bits:
               0        1 if floppy drive present (see bits 6&7)  0 if not
               1        1 if 80x87 installed  (not valid in PCjr)
               2,3      system board RAM   (not used on AT or PS/2)
                        00      16k
                        01      32k
                        10      48k
                        11      64k
               4,5      initial video mode
                        00      no video adapter
                        01      40 column color  (PCjr)
                        10      80 column color
                        11      MDA
               6,7      number of diskette drives
                        00      1 drive
                        01      2 drives
                        10      3 drives
                        11      4 drives
               8        0       DMA present
                        1       DMA not present (PCjr)
               9,A,B    number of RS232 serial ports
               C        game adapter  (joystick)
                        0       no game adapter
                        1       if game adapter
               D        serial printer (PCjr only)
                        0       no printer
                        1       serial printer present
               E,F      number of parallel printers installed
      note 1) The IBM PC and AT store the settings of the system board
              switches or CMOS RAM setup information (as obtained by the BIOS
              in the Power-On Self Test (POST)) at addresses 40:10h and
              40:13h. 00000001b indicates "on", 00000000b is "off".
           2) CMOS RAM map, PC/AT:
             offset              contents
               00h          Seconds
               01h          Second Alarm
               02h          Minutes
               03h          Minute Alarm
               04h          Hours
               05h          Hour Alarm
               06h          Day of the Week
               07h          Day of the Month
               08h          Month
               09h          Year
               0Ah          Status Register A
               0Bh          Status Register B
               0Ch          Status Register C
               0Dh          Status Register D
               0Eh          Diagnostic Status Byte
               0Fh          Shutdown Status Byte
               10h          Disk Drive Type for Drives A: and B:
                            The drive-type bytes use bits 0:3 for the first
                            drive and 4:7 for the other
                            Disk drive types:
                            00h         no drive present
                            01h         double sided 360k
                            02h         high capacity (1.2 meg)
                            03h-0Fh     reserved
               11h          (AT):Reserved    (PS/2):drive type for hard disk C:
               12h          (PS/2):drive type for hard disk D:
                            (AT, XT/286):hard disk type for drives C: and D:
                            Format of drive-type entry for AT, XT/286:
                            0       number of cyls in drive (0-1023 allowed)
                            2       number of heads per drive (0-15 allowed)
                            3       starting reduced write compensation (not
                                    used on AT)
                            5       starting cylinder for write compensation
                            7       max. ECC data burst length, XT only
                            8       control byte
                                    Bit
                                    7       disable disk-access retries
                                    6       disable ECC retries
                                    5-4     reserved, set to zero
                                    3       more than 8 heads
                                    2-0     drive option on XT (not used by AT)
                            9       timeout value for XT (not used by AT)
                            12      landing zone cylinder number
                            14      number of sectors per track (default 17,
                                    0-17 allowed)
               13h          Reserved
               14h          Equipment Byte (corresponds to sw. 1 on PC and XT)
               15h-16h      Base Memory Size      (low,high)
               17h-18h      Expansion Memory Size (low,high)
               19h-20h      Reserved
                            (PS/2) POS information Model 50 (60 and 80 use a 2k
                            CMOS RAM that is not accessible through software)
               21h-2Dh      Reserved (not checksumed)
               2Eh-2Fh      Checksum of Bytes 10 Through 20  (low,high)
               30h-31h      Exp. Memory Size as Det. by POST (low,high)
               32h          Date Century Byte
               33h          Information Flags (set during power-on)
               34h-3Fh      Reserved
           3) The alarm function is used to drive the BIOS wait function (int
              15h function 90h).
           4) To access the configuration RAM write the byte address (00-3Fh)
              you need to access to I/O port 70h, then access the data via I/O
              port 71h.
           5) CMOS RAM chip is a Motorola 146818
           6) The equipment byte is used to determine the configuration for the
              power-on diagnostics.
           7) Bytes 00-0Dh are defined by the chip for timing functions, bytes
              0Eh-3Fh  are defined by IBM.
40:12  byte    number of errors detected by infrared keyboard link (PCjr only)
40:13  word    availible memory size in Kbytes (less display RAM in PCjr)
               this is the value returned by int 12h
40:17  byte    keyboard flag byte 0 (see int 9h)
               bit 7  insert mode on      3  alt pressed
                   6  capslock on         2  ctrl pressed
                   5  numlock on          1  left shift pressed
                   4  scrollock on        0  right shift pressed
40:18  byte    keyboard flag byte 1 (see int 9h)
               bit 7  insert pressed      3  ctrl-numlock (pause) toggled
                   6  capslock pressed    2  PCjr keyboard click active
                   5  numlock pressed     1  PCjr ctrl-alt-capslock held
                   4  scrollock pressed   0
```

```
40:19  byte    storage for alternate keypad entry (not normally used)
40:1A  word    pointer to keyboard buffer head character
40:1C  word    pointer to keyboard buffer tail character
40:1E  32bytes 16 2-byte entries for keyboard circular buffer, read by int 16h
40:3E  byte    drive seek status - if bit=0, next seek will recalibrate by
               repositioning to Track 0.
               bit 3  drive D          bit 2  drive C
                   1  drive B              0  drive A
40:3F  byte    diskette motor status
               bit 7  1, write in progress 3  1, D: motor on (floppy 3)
                   6                       2  1, C: motor on (floppy 2)
                   5                       1  1, B: motor on
                   4                       0  1, A: motor on
40:40  byte    motor off counter
               starts at 37 and is decremented 1 by each system clock tick.
               motor is shut off when count = 0.
40:41  byte    status of last diskette operation     where:
               bit 7 timeout failure            bit 3 DMA overrun
                   6 seek failure                   2 sector not found
                   5 controller failure             1 address not found
                   4 CRC failure                    0 bad command
40:42  7 bytes NEC status
40:49  byte    current CRT mode (hex value)
                   00h 40x25 BW      (CGA)          01h 40x25 color   (CGA)
                   02h 80x25 BW      (CGA)          03h 80x25 color   (CGA)
                   04h 320x200 color (CGA)          05h 320x200 BW    (CGA)
                   06h 640x200 BW    (CGA)          07h monochrome    (MDA)
               extended video modes (EGA/MCGA/VGA or other)
                   08h lores,16 color               09h med res,16 color
                   0Ah hires,4 color                0Bh n/a
                   0Ch med res,16 color             0Dh hires,16 color
                   0Eh hires,4 color                0Fh hires,64 color
40:4A  word    number of columns on screen, coded as hex number of columns
               20 col = 14h  (video mode 8, low resolution 160x200 CGA graphics)
               40 col = 28h
               80 col = 46h
40:4C  word    screen buffer length in bytes
               (number of bytes used per screen page, varies with video mode)
40:4E  word    current screen buffer starting offset (active page)
40:50  8 words cursor position pages 1-8
               the first byte of each word gives the column (0-19, 39, or 79)
               the second byte gives the row (0-24)
40:60  byte    end line for cursor   (normally 1)
40:61  byte    start line for cursor (normally 0)
40:62  byte    current video page being displayed  (0-7)
40:63  word    base port address of 6845 CRT controller or equivalent
               for active display           3B4h=mono, 3D4h=color
40:65  byte    current setting of the CRT mode register
40:66  byte    current palette mask setting  (CGA)
40:67  5 bytes temporary storage for SS:SP during shutdown (cassette interface)
40:6C  word    timer counter low word
40:6E  word    timer counter high word
40:69  byte    HD_INSTALL (Columbia PCs) (not valid on most clone computers)
               bit  0 = 0  8 inch external floppy drives
                      1  5-1/4 external floppy drives
                  1,2 =    highest drive address which int 13 will accept
                           (since the floppy drives are assigned 0-3, subtract
                           3 to obtain the number of hard disks installed)
                  4,5 =    # of hard disks connected to expansion controller
                  6,7 =    # of hard disks on motherboard controller
                           (if bit 6 or 7 = 1, no A: floppy is present and
                           the maximum number of floppies from int 11 is 3)
40:70  byte    24 hour timer overflow 1 if timer went past midnight
               it is reset to 0 each time it is read by int 1Ah
40:71  byte    BIOS break flag (bit 7 = 1 means break key hit)
40:72  word    reset flag (1234 = soft reset, memory check will be bypassed)
               PCjr keeps 1234h here for softboot when a cartridge is installed
40:74  byte    status of last hard disk operation; PCjr special diskette control
40:75  byte    # of hard disks attached (0-2)    ; PCjr special diskette control
40:76  byte    hd control byte; temporary holding area for 6th param table entry
40:77  byte    port offset to current hd adapter ; PCjr special diskette control
40:78  4 bytes timeout value for LPT1,LPT2,LPT3,LPT4
40:7C  4 bytes timeout value for COM1,COM2,COM3,COM4 (0-FFh seconds, default 1)
40:80  word    pointer to start of circular keyboard buffer, default 03:1E
40:82  word    pointer to end of circular keyboard buffer, default 03:3E
40:84  byte    rows on the screen (EGA only)
40:84  byte    PCjr interrupt flag; timer channel 0  (used by POST)
40:85  word    bytes per character (EGA only)
40:85  2 bytes (PCjr only) typamatic char to repeat
40:86  2 bytes (PCjr only) typamatic initial delay
40:87  byte    mode options (EGA only)
               Bit 1   0 = EGA is connected to a color display
                       1 = EGA is monochrome.
               Bit 3   0 = EGA is the active display,
                       1 = "other" display is active.
                  Mode combinations:
                  Bit3  Bit1     Meaning
                    0     0   EGA is active display and is color
                    0     1   EGA is active display and is monochrome
                    1     0   EGA is not active, a mono card is active
                    1     1   EGA is not active, a CGA is active
40:87  byte    (PCjr only) current Fn key code
40:88  byte    feature bit switches (EGA only) 0=on, 1=off
               bit 3 = switch 4
               bit 2 = switch 3
               bit 1 = switch 2
               bit 0 = switch 1
40:88  byte    (PCjr only) special keyboard status byte
               bit 7 function flag      3 typamatic (0=enable,1=disable)
                   6 Fn-B break         2 typamatic speed (0=slow,1=fast)
                   5 Fn pressed         1 extra delay bef.typamatic (0=enable)
                   4 Fn lock            0 write char, typamatic delay elapsed
40:89  byte    PCjr, current value of 6845 reg 2 (horiz.synch) used by
               ctrl-alt-cursor screen positioning routine in ROM
40:8A  byte    PCjrCRT/CPU Page Register Image, default 3Fh
40:8B  byte    last diskette data rate selected
40:8C  byte    hard disk status returned by controller
40:8D  byte    hard disk error returned by controller
40:8E  byte    hard disk interrupt (bit 7=working int)
40:90  4 bytes media state drive 0, 1, 2, 3
40:94  2 bytes track currently seeked to drive 0, 1
40:96  byte    keyboard flag byte 3 (see int 9h)
40:97  byte    keyboard flag byte 2 (see int 9h)
```

```
40:98  dword   pointer to users wait flag
40:9C  dword   users timeout value in microseconds
40:A0  byte    real time clock wait function in use
40:A1  byte    LAN A DMA channel flags
40:A2  2 bytes status LAN A 0,1
40:A4  dword   saved hard disk interrupt vector
40:A8  dword   EGA pointer to parameter table
40:B4  byte    keyboard NMI control flags (Convertible)
40:B5  dword   keyboard break pending flags (Convertible)
40:B9  byte    port 60 single byte queue (Convertible)
40:BA  byte    scan code of last key (Convertible)
40:BB  byte    pointer to NMI buffer head (Convertible)
40:BC  byte    pointer to NMI buffer tail (Convertible)
40:BD  16bytes NMI scan code buffer (Convertible)
40:CE  word    day counter (Convertible and after)
to -04:8F              end of BIOS Data Area
40:90-40:EF            reserved by IBM
40:F0 16 bytes Intra-Application Communications Area (for use by applications
40:FF          to transfer data or parameters to each other)
50:00  byte    DOS print screen status flag
                       00h    not active or successful completion
                       01h    print screen in progress
                       0FFh   error during print screen operation
50:01          Used by BASIC
50:02-03       PCjr POST and diagnostics work area
50:04  byte    Single drive mode status byte
                       00     logical drive A
                       01     logical drive B
50:05-0E       PCjr POST and diagnostics work area
50:0F          BASIC: SHELL flag (set to 02h if there is a current SHELL)
50:10  word    BASIC: segment address storage (set with DEF SEG)
50:12  4 bytes BASIC: int 1Ch clock interrupt vector segment:offset storage
50:16  4 bytes BASIC: int 23h ctrl-break interrupt segment:offset storage
50:1A  4 bytes BASIC: int 24h disk error interrupt vector segment:offset storage
50:1B-1F       Used by BASIC for dynamic storage
50:20-21       Used by DOS for dynamic storage
50:22-2C       Used by DOS for diskette parameter table. See int 1Eh for values
50:30-33       Used by MODE command
50:34-FF       Unknown - Reserved for DOS
0008:0047 IO.SYS or IBMBIO.COM IRET instruction. This is the dummy routine that
          interrupts 01h, 03h, and 0Fh are initialized to during POST.
```

Danach ist es zunächst einmal aus mit absolut spezifizierbaren Adreßinhalten. Den folgenden Bereich, bis hinauf zur Adresse `09fff` verwaltet das jeweilige Betriebssystem, in unserem Fall MSDOS. Danach folgen Video-Adapter mit wechselnden Inhalten, siehe Tabelle in Folge 2.

Upper memory

Die folgenden Adressen sind jetzt nicht mehr so ganz verläßlich.

```
C000:001E EGA BIOS signature (the letters IBM)
F000:E05B loc Reset
F000:E2C3 loc NMI Entry Point
```

Hard Disk Information Tables

Jeder Tabelleneintrag enthält 16 Bytes für jede Festplattentype. Diese Tafel kann mit BIOS-Hersteller und BIOS-Datum anders aufgebaut sein; die hier abgebildete hat als ersten Eintrag 0.

```
F000:E331 dw hdsk_cylinders Number of cylinders, hdsk_type_0
F000:E333 db hdsk_heads Number of heads
F000:E334 dw hdsk_lo_wrt_cyl Low write current cyl begin (XT)
F000:E336 dw hdsk_precompcyl Write pre-compensation cylinder
F000:E338 db hdsk_err_length Error correction burst length (XT)
F000:E339 db hdsk_misl_bits
Miscellaneous bit functions:
bits 0-2 disk option, XT only (XT)
     0-2 unused, all others
     3 = 1 if > 8 heads
     4   unused
     5 = 1 for bad map at last cylinder + 1
     6 or 7 = 1 no retries
F000:E33A db hdsk_timeout Normal timeout  (XT)
F000:E33B db hdsk_fmt_timout Format timeout  (XT)
F000:E33C db hdsk_chk_timout Check timeout   (XT)
F000:E33D dw hdsk_parkng_cyl Parking cylinder number
F000:E33F db hdsk_sectr_trac Number of sectors per track
F000:E340 db hdsk_unused Unused
F000:E331 ds hdsk_type_
F000:E6F2 loc Bootstrap Load
System Configuration Table
F000:E6F5 dw Config_tbl_size  Size of table in bytes
F000:E6F7 db Config_model Model type
  0F8h = 80386 model 70-80 types
  0FCh = 80286 model 50-60 types, also most 80286/80386 compatibles
  0FAh = 8088/86 model 25-30 type
F000:E6F8 db Config_sub_model Sub-Model type
F000:E6F9 db Config_BIOS_rev BIOS revsion number
F000:E6FA db Config_features Feature information
  bit 7=1, hard disk uses DMA 3
  bit 6=1, dual interrupt chips
  bit 5=1, has real-time-clock
  bit 4=1, int 15h, ah=4Fh is supported (keyboard)
  bit 3=1, external wait support
  bit 2=1, has extended BIOS RAM
  bit 1=1, micro-channel
  bit 0=1, unused
F000:E6FB db Config_info_bytes Information bytes (future use)
Baud Rate Table
Table of hex divsors  for the serial ports. Table divisors for bauds 110 to 19,200.
F000:E729 dw baud_110, baud_rate_tbl
F000:E72B dw baud_150
F000:E72D dw baud_300
F000:E72F dw baud_600
F000:E731 dw baud_1200
F000:E733 dw baud_2400
F000:E735 dw baud_4800
F000:E737 dw baud_9600
F000:E739 dw baud_19200
F000:E82E loc Keyboard Function Call
F000:E987 loc Keyboard Hardware Interrupt
F000:EC59 loc Floppy Disk Function Call
F000:EF57 loc Floppy Disk ISR
Floppy Disk Parameters
F000:EFC7 db dsk_info_1 Start of ROM BIOS data areas
  hi nibble = stepping rate in ms
  lo nibble = head unload time, ms
F000:EFC8 db  dsk_info_2 2nd info byte bit 0 = 0 for DMA
F000:EFC9 db  dsk_motor_delay
                 Delay after use for motor off
F000:EFCA db  dsk_sectr_bytes
Bytes per sector  0 =  128 bytes
                  1 =  256 bytes
                  2 =  512 bytes
                  3 = 1024 bytes
F000:EFCB db dsk_sector_trac Number of sectors per track
F000:EFCC db dsk_head_gap Gap Length
F000:EFCD db dsk_data_length Data Length
F000:EFCE db dsk_format_gap Format Gap Length
F000:EFCF db dsk_format_byte Format write byte
F000:EFD0 db dsk_settlg_time Head load time, in ms
F000:EFD1 db dsk_startup_tim Motor startup wait time, 125ms
F000:EFD2 LPT-Fuction-Call
F000:F065 Video Function Call
Video Hardware Registers
F000:F0A4 db video_hdwr_tbl1 mode CGA 40 columns x 25 lines
F000:F0B4 db video_hdrw_tbl2 mode CGA 80 columns x 25 lines
F000:F0C4 db video_hdwr_tbl3 mode CGA graphics
F000:F0D4 db video_hdwr_tbl4 mode MDA 80 columns x 25 lines
F000:F0E4 dw video_buf_size1 Video buffer bytes CGA 40x25
F000:F0E6 dw video_buf_size2 Video buffer bytes CGA 80x25
F000:F0E8 dw video_buf_size3 Video buffer bytes CGA Graphics
F000:F0EA dw video_buf_size4 Video buffer bytes CGA Graphics
F000:F0EC db video_columntbl Video columns per modes 0-7
F000:F0F4 db video_hdwr_mode Video hardware modes (0-7)
F000:F841 loc Memory size Function call
F000:F84D loc Equipment Check Function call
F000:F859 loc Cassette Function Call
F000:FA6E db video_char_tbl Video characters in graphic modes
F000:FE6E loc Timer Function  Call
F000:FEA5 loc Timer Hardware Interrupt
F000:FEF3 dw int_vec_table Initial interrupt vectors
F000:FF1D dw int_data_table
F000:FF21 dw video_ptr
F000:FF23 dw int_vec_table_2
F000:FF53 loc Dummy Interrupt return
F000:FF54 loc Print Screen Function Call
F000:FFF0 loc power_on_reset SYSTEM RESET
```

Kalt- oder Warmstart-Einsprung

Ein Kaltstart enthält neben dem Test der CPU, des ROM und der Initialisierung der Hardware auch den Test der folgenden Komponenten:

- Memory system
- Timer/Counter (which is also used for RAM refresh)
- Interrupt Controller(s)
- DMA Controller(s)
- Keyboard Controller
- Video Controller & Video RAM
- Floppy Controller
- Hard Disk Controller (if present)

Fehler werden entweder durch Töne oder Fehlerkodes am Bildschirm angezeigt.

Ein Warmstart benutzt den Inhalt des warm_boot_flag um den Speichertest zu überspringen (i.a. beim Drücken von Ctrl-Alt-Del).

```
F000:FFF5 BIOS release date
F000:FFFE PC model identification
            date      model byte            submodel byte    revision
          04/24/81     FF = PC-0 (16k)          --              --
          10/19/81     FF = PC-1 (64k)          --              --
          08/16/82     FF = PC, XT, XT/370      --              --
                            (256k motherboard)
          10/27/82     FF = PC, XT, XT/370      --              --
                            (256k motherboard)
          11/08/82     FE = XT, Portable PC     --              --
                            XT/370, 3270PC
          01/10/86     FB = XT                  00              01
          01/10/86     FB = XT-2 (early)
          05/09/86     FB = XT-2 (640k)         00              02
          06/01/83     FD = PCjr                --              --
          01/10/84     FC = AT                  --              --
          06/10/85     FC = AT                  00              01
          11/15/85     FC = AT                  01              00
          04/21/86     FC = XT/286              02              00
          09/13/85     F9 = Convertible         00              00
          09/02/86     FA = PS/2 Model 30       00              00
          11/15/86     FC = AT, Enhanced 8mHz
          02/13/87     FC = PS/2 Model 50       04              00
          02/13/87     FC = PS/2 Model 60       05              00
          1987         F8 = PS/2 Model 80       00              00
                       2D = Compaq PC (4.77)    --              --
                       9A = Compaq Plus (XT)    --              --
                00FC 7531/2 Industrial AT
              06FC 7552 Gearbox
F000:FFFF db  model_sub_type
```

BIOS-Erweiterungen

Das System überprüft den Speicher hinsichtlich installierter ROMs in 2k-Abständen beginnend bei 0C0000H. BIOS-ROMs beginnen mit dem Code AA55h. Danch folgt ein Längenfeld und danach der eigentliche Einsprungpunkt.

Externe ROM-Module können zwischen den Adressen c800:0000 und e000:0000 vorkommen. Jeder 2K-Block in diesem Adreßbereich wird für folgende ROM-Signatur abgesucht. :

```
Offset Size Contents
            Ú---¿
 +0      1  ³55H³         Signature of BIOS-accessible ROM module
            Ã---´          (first word in segment is aa55H)
 +1      1  ³aaH³
            Ã---´
 +2      1  ³len³         length of ROM module in 512-byte increments
            Ã---Â-Â---¿
 +3      ?  ³  ³  ³executable code
            Ã---Â-Â---Ù (often a NEAR jump to initialization code)
            |  |        (a dummy byte usually exists to validate checksum)
```

Wurde eine gültige ROM-Signatur gefunden, wird diese BIOS-Erweiterung mit einem far-CALL auf Adresse 3 des ROM initialisiert. Üblicherweise initialisiert das BIOS alle erforderlichen Interruptvektoren.

Lesen aus dem Hauptspeicher,

HC05MM2.CPP, HC05MM2.C

Das folgende Beispiel zeigt, wie man aus dem Bildschirmspeicher liest, wie man sich den Wert eines Interruptvektors ansieht und wie man die Adresse der seriellen Schnittstelle erfährt.

```
/* HC05MM2.CPP */
/*
 * Lesen im Hauptspeicher
 * ======================
 */
#include <dos.h>
#include <mytypes.h>
#define P_POINTER
#define P_STREAMS
#include <portable.h>

VOID main(VOID)
{
  UINT far * uifp;
  UINT seg,off,ser;

  constream con;
  con.clrscr();

  con << "Lesen innerhalb des Hauptspeichers\n";
  uifp = (UINT far *)MK_FP(0xb800,0x0000);
  con << "Zeichen links oben am Bildschirm: "
      << hex << setw(2) << (UINT)*uifp << ':'
      << setw(1) << (UCHAR)*uifp
      << "  Attribut: "
      << hex << setw(2) << (UINT)((*uifp)>>8)
      << endl;

  uifp = (UINT far *)MK_FP(0x0000,5*4);
  off = *uifp;
  uifp++;
  seg = *uifp;
  con << "Print-Screen Interruptvektor: "
      << hex << setw(4) << seg << ':' << off << endl;

  uifp = (UINT far *)MK_FP(0x0040,0x0000);
  ser = *uifp;
  con << "Adresse der seriellen Schnittstelle: "
      << hex << setw(4) << ser
      << endl;

  con << "Ende mit Taste";
  cin.get();
}
```

Uhrzeit aus dem Speicher,

HC05MM3.CPP, HC05MM3A.CPP, HC05MM3B.CPP, HC05MM3C.CPP

Jeder einlangende Timer-Interrupt zählt die Speicherstelle 0040:006c um 1 hoch, beginnt dabei um Mitternacht mit 0. Daherhat man auf 6c und den folgenden drei Adressen eine Zahl vom C-typ `long`, die eine Maßzahl für die aktuelle Uhrzeit ist. Die Einheit ist 53ms. Das Programm HC05MM3.CPP gibt die Uhrzeit in Vielfachen von 53ms aus. Die entstehende Zahl ist nur für Differenzmessungen, nicht aber für die Auswertung in Stunden, Minuten, Sekunden geeignet.

```
/* HC05MM3.CPP */
/*
 * Lesen der Uhrzeit aus den BIOS-Daten
 * ====================================
 */
#include <dos.h>
#define P_STREAMS
#include <portable.h>
#include <mytypes.h>

VOID main(VOID)
{
  ULONG far *time_p;
  constream con;
  con.clrscr();
  cout << "Uhrzeit aus dem BIOS-Datenbereich\n";
  time_p = (ULONG far *)MK_FP(0x0040,0x006c);
  do
  {
    cout << *time_p << ' ';
    delay (1000);
  }
  while (kbhit()==0);
  cin.get();
}
```

Natürlich kann die Uhrzeit auch gleich richtig lesbar ausgegeben werden. Das Programm HC05MM3A.CPP berücksichtigt das Dastellungsformat und gibt in Stunden, Minuten und Sekunden aus:

```
/* HC05MM3A.CPP */
/*
 * Lesen der Uhrzeit aus den BIOS-Daten
 * ====================================
 */
#include <dos.h>
#define P_STREAMS
#include <portable.h>
#include <mytypes.h>

VOID main(VOID)
{
  ULONG far *time_p;
  UINT s;

  constream con;
  con.clrscr();
  cout << "Uhrzeit aus dem BIOS-Datenbereich\n";
  time_p = (ULONG far*)MK_FP(0x0040,0x006c);
  do
  {
    s = (UINT)((FLOAT)(*time_p)*0.053);
    cout << s/3600 << ':'
         << s/60 << ':'
         << s%3600 << ' ';
    delay (1000);
  }
  while (kbhit()==0);
  cin.get();
}
```

In beiden Programmen erfolgt der Hauptspeicherzugriff über Pointer, die zuerst gebildet werden müssen:

```
time_p=(ULONG far *)MK_FP(0x0040,0x006c);
```

und auf deren Ziel mit `*time_p` zugegriffen wird. Einfacher wird der Zugriff durch eine eigene Klasse, die das Arbeiten mit Pointern erspart (HC05MM3B.CPP, HC05MM3C.CPP, siehe später).

So wie diese gezeigten Beispiele gibt es viele Adressen, deren Wert uns Aufschluß über den Zustand des Rechners gibt.

Darüber wollen wir uns jetzt eine Übersicht verschaffen:

Nachdem wir einige dieser Adressen exemplarisch analysiert haben, schreiben wir zwei Hilfsprogramme, die eine vollständige, wenn auch nicht dechifrierte, Liste der BIOS-Variablen am Bildschirm ausgeben.

Verzeichnis der BIOS-Variablen, HC05MM5.CPP

Die BIOS-Variablen können gleichzeitig auf einem Bildschirm angezeigt werden. Um sie zu interpretieren, benötigt man noch eine Tabelle, wie etwa jene im selben Heft; noch besser wäre es, die Texte aus der Erklärung der BIOS-Daten mit den Daten selbst zu verknüpfen, wie das auch bei verschiedenen Systemanalyseprogrammen geschieht.

Beiden Programmen gemeinsam ist die Darstellung der Daten in Zeilen und Spalten. Die Konstante `COLW` legt die Spaltenbreite, `COL` die Spaltenzahl und `ROW` die Zeilenzahl. Die jeweilige Position wird in zwei Variablen `row` und `col` gehalten, die durch `col=(num/ROW)%COL` und `row=num%ROW` berechnet werden. Die Kursorposition für den Schreibbeginn findet man mit `setxy(1+col*COLW,row+2)`, wobei 1 und 2 feste Abstände vom Rand darstellen.

```
/* HC05MM5.CPP */
/*
 * Anzeige der BIOS-Daten
 * ======================
 */
#include <dos.h>
#include <mytypes.h>
#define P_STREAMS
#define P_POINTER
#include <portable.h>

#define COLW 4
#define COL 16
#define ROW 16

VOID main(VOID)
{
  INT off, row, col;
  UCHAR far * ip;

  constream con;
  con.clrscr();
  con << "BIOS-Daten\n";

  ip=(UCHAR far *)MK_FP(0x40,0);

  for (off=0; off<0x10; off++) // niederwertiges byte
  {
    con << setxy(1,off+4) << hex << setfill('0')
        << off << " |";
  }

  for (off=0; off<0x10; off++) // höherwertiges byte
  {
    con << setxy((off*COLW)+5,2)
        << hex << setfill('0') << off << " |";
    con << setxy((off*COLW)+5,3) << "----";
  }

  for (off=0; off<0x100; off++)
  {
    col=(off/ROW)%COL;
    row=off%ROW;
    con << setxy(5+col*COLW,row+4)
        << hex << setfill('0') << setw(2)
        << (UINT)*ip;
    ip++;
  }
  con << setxy(1,wherey()+1);
  con << "Taste";
  cin.get();
}
```

Nachdem wir jetzt die Namen der Interruptvektoren kennen, ihre Adresse, dann auch eine Liste der BIOS-Variablen kennen, und auch ein geeignetes Ausdruckprogramm dafür besitzen, wäre es eine lohnende Aufgabe, die Daten auch mnemonisch auszuwerten, ähnlich, wie es die bekannten Analyseprogramme, wie `MFT` (=Manifest/Quarterdek), `SYSINFO` (=System Information/Norton), `MSD` (=Microsoft Diagnostics/Microsoft) oder `CHECKIT`() tun. Gefragt ist also eine Liste der Interruptvektoren, wobei als Zusatzinformation eine leicht erkennbare Kurzbezeichnung mitangegeben wird. Genauso soll eine BIOS-Datenliste ausgegeben werden, wobei die Daten ihrer korrekten Länge und Gewichtung entsprechend gereiht werden.

Austausch der seriellen und parallelen Ports, HC05MM6.C

Interessanterweise werden sogar noch bei der DOS-Version 6.0 verschiedene kleine aber für den Betrieb eines Rechners doch wichtige Manipulationen nicht unterstützt. Es sind zwar drei LPTs vorhanden, die Standardausgabe erfolgt auf `LPT1`. Will man die Standardausgabe z.B. auf `LPT2` ändern, muß man die entsprechenden Adressen im BIOS-Datenteil austauschen; etwas, das ohne hardwarenahe Programmierung nicht lösbar ist, da es weder durch das BIOS noch durch das Betriebssystem unterstützt wird.

Das Programm hieß `portex.c` und wurde für diese Serie in `HC05MM6.C` umbenannt. Der Hilfetext im Programm wurde nicht geändert.

Das Programm `PORTEX` erlaubt es die Adressen von seriellen oder parallelen Schnittstellen auszutauschen. Diese Möglichkeit ist im Betriebssystem nicht vorgesehen, daher muß man es mit geeigneten Utilities - wie diesem - ermöglichen.

Was in einem PC LPT1 (oder COM1) ist, bestimmt das BIOS beim Initialisieren des BIOS-Datenbereichs. Leider gibt es hier eine Unklarheit bei der Namensgebung: einerseits gibt es in einem System eine bis mehrere parallele und serielle Schnittstellen. Bezeichnen wir diese mit LPT1h...LPT3h, COM1h..COM4h. Anderseits gibt es für diese Kanäle je einen Speicherplatz im BIOS-Datenbereich.

Der MSDOS-Druckkanal `PRN` wird `LPT1` zugewiesen. Jedes Mitprotokollieren mit ^P oder jede Druckerausgabe in C an `stdprn` wird an `LPT1` gesendet. MSDOS erfährt über die vom BIOS initialisierten Adressen, was `LPT1` ist. So wie DOS sollte es auch jedes andere 'saubere' Programm machen: nicht die Hardware selbst an 0x3bc ansprechen, sondern über die MSDOS-Druckausgabe (MSDOS-Funktion 5). Unglaubwürdig wird dieses Verfahren, wenn man an einen anderen Kanal ausgeben will, z.B. `LPT2`. Dann muß man dafür sorgen, daß MSDOS die Adresse von LPT2 schreibt; aber wie, wenn es nur einen Kanal, `PRN` gibt?

`PORTEX` erwartet in der Kommandozeile die Angabe zweier Argumente, die Ports, die es auszutauschen gilt. Erlaubt sind `LPT1`, `LPT2`, `LPT3`, `COM1`, `COM2`, `COM3`, `COM4`. Jede unerlaubte Kombination wird durch die große `if..else if..`-Kombination am Eingang abgefangen und ein entsprechender Fehlertext ausgegeben. Für eine saubere Trennung von Text und Kode wird eingangs ein Array von Meldetexten in Form von `errtxt[]` angelegt. Zur Vereinfachung der Anzeige wird der Text nicht direkt über `printf()` angezeigt, sondern über die Funktion `error()`; damit wird erreicht, daß alle Anzeigen gleichartig erfolgen und jede Änderung des Ausgabeformats sich auf alle Ausgaben auswirkt.

Der Austauschvorgang selbst wird mit der Funktion `swap()` bewerkstelligt, die zwei Pointer auf die auszutauschenden Speicherstellen als Parameter übernimmt.

Bei einem Fehler in der Kommandozeile oder wenn die Kommandozeile keinen Parameter enthält, wird die Funktion `display_status()` aufgerufen, die die aktuellen Adressen anzeigt. `display_status()` wird auch vor und nach Autausch der Portadressen ausgeführt.

```
/* HC05MM6.C */
/*
 * Austausch serieller und paralleler Ports
 * ========================================
 */
#include <string.h>
#include <stdio.h>
#include <dos.h>
#include <mytypes.h>

VOID swap(UINT far *i, UINT far *j);
VOID display_status(VOID);
VOID error(CHAR *txt);

CHAR *errtxt[] = {
  "Invalid number of arguments",
  "Incorrect length in argument 1",
  "Incorrect length in argument 2",
  "Arguments must have equal type",
  "Port-number of arg1 must be in the Range of 1..4",
  "Port-number of arg2 must be in the Range of 1..4",
  "There is nothing to do",
  "Port-number of arg1 must be in the Range of 1..3",
  "Port-number of arg2 must be in the Range of 1..3",
  "There is nothing to do",
  "Only LPTx and COMx are allowed as arg1",
    "\nExchange of Ports\n"
    "=================\n"
    "Usage:\n"
    "portex                         Status-Display\n"
    "portex ?                       This display\n"
    "portex LPTx LPTy  x,y={1,2,3}  Exchange LPTs \n"
    "portex COMx COMy  x,y={1,2,3,4} Exchange COMs \n" };

VOID main(INT argc, CHAR * argv[])
{
  UINT far *fp;
  UINT far *fq;

  if (argc==1) // Status-Anzeige der Ports
  {
    display_status();
    return;
  }
  else if ((argc==2) && (argv[1][0]=='?'))
  {
  }
  else if (argc!=3)
  {
    error(errtxt[0]);
  }
  else if (strlen(argv[1])!=4)
  {
    error(errtxt[1]);
  }
  else if (strlen(argv[2])!=4)
  {
    error(errtxt[2]);
  }
  else if (strnicmp(argv[1],argv[2],3)!=0)
  {
    error(errtxt[3]);
  }
  else if (strnicmp(argv[1],"COM",3)==0) // change COM-Ports
  {
    if ((argv[1][3]<'1') || (argv[1][3]>'4'))
    {
      error(errtxt[4]);
    }
    else if ((argv[2][3]<'1') || (argv[2][3]>'4'))
    {
      error(errtxt[5]);
    }
    else if (argv[1][3]==argv[2][3])
    {
      error(errtxt[6]);
    }
    else
    {
      fp=(UINT far *)MK_FP(0x0040,(argv[1][3]-'1')*2);
      fq=(UINT far *)MK_FP(0x0040,(argv[2][3]-'1')*2);
      printf("Before: ");
      display_status();
      swap(fp,fq);
      printf("After:  ");
      display_status();
      return;
    }
  }
  else if (strnicmp(argv[1],"LPT",3)==0) // change LPT-Ports
  {
    if ((argv[1][3]<'1') || (argv[1][3]>'3'))
    {
      error(errtxt[7]);
    }
    else if ((argv[2][3]<'1') || (argv[2][3]>'3'))
    {
      error(errtxt[8]);
    }
    else if (argv[1][3]==argv[2][3])
    {
      error(errtxt[9]);
    }
    else
    {
      fp=(UINT far *)MK_FP(0x0040,(argv[1][3]-'1')*2 + 8);
      fq=(UINT far *)MK_FP(0x0040,(argv[2][3]-'1')*2 + 8);
      printf("Before: ");
      display_status();
      swap(fp,fq);
      printf("After:  ");
      display_status();
      return;
    }
  }
  else
  {
    error(errtxt[10]);
  }
  error(errtxt[11]);
}

VOID swap(UINT far *i, UINT far *j)
{
  UINT temp;
  temp=*j; *j=*i; *i=temp;
}

VOID display_status(VOID)
{
  UINT far *fp;
  CHAR * txt[] = { "COM1","COM2","COM3","COM4",
                   "LPT1","LPT2","LPT3" };
  INT i;

  for (i=0; i<7; i++)
  {
    fp=(UINT far *)MK_FP(0x0040,i*2);
    if (*fp)
      printf("%s:%3x ",txt[i],*fp);
    else
      printf("%s:--- ",txt[i],*fp);
  }
  printf("\n");
}

VOID error(CHAR *txt)
{
  printf("%s!\n\n",txt);
}
```

Die Liste der nützlichen Programme, die die absoluten Speicheradressen benutzen, könnte noch weiter fortgesetzt werden. Wir wollen uns aber jetzt einen allgemeineren Standpunkt für den Speicherzugriff überlegen.

Klassen M,MEM,BIOS,IBIOS

Der Speicherzugriff erfolgt immer auf folgende Weise:

Pointer generieren: `TYP far * t = (TYP far *)MK_FP(seg,off);`
Wert bearbeiten: `*t=5;`

Nachteile:

(1) man muß mit den dereferenzierten Werten arbeiten (`*t`), es gibt keine Variable für diesen Speicherplatz.
(2) der Pointer ist gegen Veränderungen nicht abgesichert.

C++ bietet die Möglichkeit, in einer eigenen Memory-Klasse absolute Speichervariable zu verwalten. Wie, entnehmen Sie aus der folgenden Headerdatei `memabs.h`, die auch den gesamten Kode innerhalb der Klasse enthält.

Klassenbaum

MEM
3
ÃÄMEM_P
3
ÀÄBIOS
3
ÀÄIBIOS

Die Inhalte der auf absoluten Speicheradressen abgelegten Variablen haben verschiedene Länge und verschiedene Bedeutung. Im Terminus von C sind es die Typen `char, int, long` sowie pointer auf diese Typen oder Arrays davon. Eine Klasse im herkömmlichen Sinn, die einen solchen Speicherplatz beschreibt, hätte den Nachteil, daß sie für die anderen Typen noch einmal formuliert werden müßte, obwohl die Inhalte der Klasse im Prinzip dieselben wären. Die neue 'Ausbaustufe' von C++, Version 2.1 von ATT, bietet dafür Abhilfe in Form sogenannter 'templates', das sind Klassen oder Funktionen, bei denen ein oder mehrere Variablentypen noch nicht feststehen aber der Kode typenunabhängig verfaßt werden kann. Diese Konzeption bietet sich hier an. Die Basisklasse `M`

Vorbemerkung zu `MEM`...

Der schnellste Zugriff auf absolute Speicheradressen ist nach wie vor der zuvor beschriebene. Kommt es aber weniger auf die Schnelligkeit, sondern eher auf einen verständlichen Kode an, kann man die folgenden Klassen `MEM`... benutzen.

Der Phantasie sind bei der Konstruktion von Klassen keine Grenzen gesetzt, man kann sie, wie diese Beispiele zeigen, auch erfolgreich für hardwarenahes Programmieren einsetzen. Ob sich diese Konstruktionen bewähren, ob man sie in dieser Form einsetzen kann, wird die Kompiler- entwicklung zeigen. Derzeit bietet nur der BORLAND-Kompiler Templates an. Die Kompiler für Mikrokontroller kennen noch nicht einmal C++. Ob sie es je lernen werden? Wir werden sehen!

Die Klasse `MEM` hat die Aufgabe eine absolute Speicheradresse zuverwalten. Nicht einfach nur ein Byte, sondern auch 2 oder 4 Bytes, je nach Bedarf. Dazu wird die Klasse `MEM`, wie auch alle folgenden Klassen, um einen fiktiven Typ `T` konstruiert, der erst bei der Definition eines Objekts dieser Klassen eingesetzt wird, je nachdem, welchen Typ die absolute Speicherstelle enthält.

`MEM` besitzt zwei Variablen: `Val` und `p`. `p` ist ein `far`-Pointer auf die gewünschte absolute Speicherstelle und `Val` ist der aus dieser Speicherstelle zuletzt gelesene Wert. Bei den später verfaßten Klasse `MEM_P` hat es sich als praktisch erwiesen, sich den letzten Wert innerhalb der Klasse, `Old` zu merken.

Es gibt vier Konstruktoren für MEM:

```
MEM(T far *tp)      // Mit einem Pointer
MEM(UINT s, UINT o) // Mit Segment und Offset
MEM(T far *tp, T t) // Mit einem Pointer und
                    // einem Initialisierungswert
MEM(UINT s, UINT o) // Mit Segment und Offset und
                    // einem Initialisierungswert
```

Vier Meldefunktionen geben über den Zustand der Klasse Auskunft: `val()` liefert den zuletzt gelesenen Wert, `addr()` liefert den Wert des Pointers, `seg()` liefert das Segment und `off()` den Offset.

Die Funktion `operator=()` dient zum Initialisieren der absoluten Adresse, die Funktion `operator()()` liefert den aktuellen Wert der absoluten Adresse.

Die absolute Adresse kann mit den Operatorfunktionen

`++,--,<<,>>,~,+=,-=,*=,/=,%=,&=,|=,^=`

so wie jede andere Variable auch bearbeitet werden.

Ein Objekt der Klasse `MEM` wird etwa wie folgt angewendet:

Die absolute Adresse `bildanfang` zeigt auf das Zeichen links oben am Bildschirm.

```
MEM<UINT> bildanfang(0xb800,0);
```

Die Operatorfunktion `operator()` liefert den aktuellen Wert.

```
cout << "Links oben steht: " << hex << bildanfang() << endl;
```

Die absolute Adresse `bildanfang` kann wie alle anderen eingebauten Typen bearbeitet werden:

```
bildanfang++;
```

Jetzt steht links oben ein anderes Zeichen:

```
cout << "Links oben steht: " << hex << bildanfang() << endl;
```

Die Klasse `MEM_P` ist von `MEM` abgeleitet. Die adressierte absolute Speicherstelle enthält einen `far`-Pointer. `MEM_P` verwaltet nicht den Zahlenwert auf einer absoluten Adresse, sondern den Wert jener Adresse auf die der Pointer zeigt, der auf `MEM_P` gespeichert ist. Neben dem indirekten Zugriff ist auch eingebaut, daß der jeweils zuletzt gelesene Wert in `Old` gespeichert wird. Damit im Zusammenhang ist auch die Funktion `changed()`, die meldet, ob eine Zustandsänderung seit der letzten Befragung der Adresse stattgefunden hat.

Es hat sich als erforderlich erwiesen, zusätzlich zu den Elemetfunktionen `seg()`, `off()` und `addr()` auch noch `pointseg()`, `pointoff()` und `pointaddr()` einzusetzen, die den Wert des Pointers zurückmelden.

MEMABS.H

```
#ifndef __MEMABS_HPP
#define __MEMABS_HPP

#define P_POINTER
#include <portable.h>
#include <mytypes.h>

#define GETMEM() Val=operator()()
#define SETMEM() operator=(Val);return Val

template <class T>
class MEM
{
protected:

  T Val;
  T far *p;

public:

  MEM(T far *tp)
    { p=tp; operator()(); }
  MEM(UINT s, UINT o)
    { p=(T far *)MK_FP(s,o); operator()(); }
  MEM(T far *tp, T t)
    { p=tp; operator=(t); }
  MEM(UINT s, UINT o, T t)
    { p=(T far *)MK_FP(s,o); operator=(t); }

  virtual VOID operator = (T t)
    { Val = *p = t; }
  virtual T operator () ()
    { Val = *p; return Val; }
```

```
  // Ueberladene Operatoren
  T operator ++ ()
    { GETMEM(); Val++; SETMEM(); }
  T operator -- ()
    { GETMEM(); Val--; SETMEM(); }
  T operator << (UINT i)
    { GETMEM(); Val<<=i; SETMEM(); }
  T operator >> (UINT i)
    { GETMEM(); Val>>=i; SETMEM(); }
  T operator ~ ()
    { GETMEM(); Val=~Val; SETMEM(); }
  T operator += (T t)
    { GETMEM(); Val+=t; SETMEM(); }
  T operator -= (T t)
    { GETMEM(); Val-=t; SETMEM(); }
  T operator *= (T t)
    { GETMEM(); Val*=t; SETMEM(); }
  T operator /= (T t)
    { GETMEM(); Val/=t; SETMEM(); }
  T operator %= (T t)
    { GETMEM(); Val%=t; SETMEM(); }
  T operator &= (T t)
    { GETMEM(); Val&=t; SETMEM(); }
  T operator |= (T t)
    { GETMEM(); Val|=t; SETMEM(); }
  T operator ^= (T t)
    { GETMEM(); Val^=t; SETMEM(); }

  T val() { return Val; }
  T far *addr() { return p; }

  UINT seg() { return FP_SEG(p); }
  UINT off() { return FP_OFF(p); }
};

template <class T>
class MEM_P : public MEM<T>
{
public:

  MEM_P(UINT s, UINT o) : MEM<T>(s,o)
    { Old = MEM<T>::operator()(); }
  MEM_P(UINT seg, UINT off, T t) : MEM<T>(seg,off,t)
    { MEM<T>::operator=(t); Old = t; }

  VOID operator = (T t)
   { Old = Val;
     **((T far **)p) = t;
     Val = t;
   }

  T operator () ()
    { Old = Val;
      Val = **((T far **)p);
      return Val;
    }

  BOOL changed()
    { return (Old==Val) ? FALSE : TRUE; }
  UINT pointoff()
    { return FP_OFF((T far *)(*p)); }
  UINT pointseg()
    { return FP_SEG((T far *)(p)); }
  T far *pointaddr()
    { return (T far *)(*p); }

protected:

  T Old;
};

#endif // __cplusplus

typedef enum BIOSA
{
  rs232_port_1       = 0x4100, // UINT
  rs232_port_2       = 0x4102, // UINT
  rs232_port_3       = 0x4104, // UINT
  rs232_port_4       = 0x4106, // UINT
  prn_port_1         = 0x4108, // UINT
  prn_port_2         = 0x410A, // UINT
  prn_port_3         = 0x410C, // UINT
  BIOS_data_seg      = 0x410E, // UINT
  equip_bits         = 0x4110, // UINT
  init_test_flag     = 0x4212, // UCHAR
  main_ram_size      = 0x4113, // UINT
  chan_io_size       = 0x4115, // UINT
  keybd_flags_1      = 0x4217, // UCHAR
  keybd_flags_2      = 0x4218, // UCHAR
  keybd_alt_num      = 0x4219, // UCHAR
  keybd_q_head       = 0x411A, // UINT
  keybd_q_tail       = 0x411C, // UINT
  keybd_queue        = 0x491E, // UINT
  dsk_recal_stat     = 0x423E, // UCHAR
  dsk_motor_stat     = 0x423F, // UCHAR
  dsk_motor_tmr      = 0x4240, // UCHAR
  dsk_ret_code       = 0x4241, // UCHAR
  dsk_status_1       = 0x4242, // UCHAR
  dsk_status_2       = 0x4243, // UCHAR
  dsk_status_3       = 0x4244, // UCHAR
  dsk_status_4       = 0x4245, // UCHAR
  dsk_status_5       = 0x4246, // UCHAR
  dsk_status_6       = 0x4247, // UCHAR
  dsk_status_7       = 0x4248, // UCHAR
  video_mode         = 0x4249, // UCHAR
  video_columns      = 0x414A, // UINT
  video_buf_siz      = 0x414C, // UINT
  video_segment      = 0x414E, // UINT
  vid_curs_pos0      = 0x4150, // UINT
  vid_curs_pos1      = 0x4152, // UINT
  vid_curs_pos2      = 0x4154, // UINT
  vid_curs_pos3      = 0x4156, // UINT
  vid_curs_pos4      = 0x4158, // UINT
  vid_curs_pos5      = 0x415A, // UINT
  vid_curs_pos6      = 0x415C, // UINT
  vid_curs_pos7      = 0x415E, // UINT
  vid_curs_mode      = 0x4160, // UINT
  video_page         = 0x4262, // UCHAR
  video_port         = 0x4163, // UINT
  video_mode_reg     = 0x4265, // UCHAR
  video_color        = 0x4266, // UCHAR
  gen_io_ptr         = 0x4167, // UINT
  gen_io_seg         = 0x4169, // UINT
  gen_int_occured    = 0x426B, // UCHAR
  timer_low          = 0x416C, // UINT
  timer_hi           = 0x416E, // UINT
  timer_rolled       = 0x4270, // UCHAR
  keybd_break        = 0x4271, // UCHAR
  warm_boot_flag     = 0x4172, // UINT
  hdsk_status_1      = 0x4274, // UCHAR
  hdsk_count         = 0x4275, // UCHAR
  hdsk_head_ctrl     = 0x4276, // UCHAR
  hdsk_ctrl_port     = 0x4277, // UCHAR
  prn_timeout_1      = 0x4278, // UCHAR
  prn_timeout_2      = 0x4279, // UCHAR
  prn_timeout_3      = 0x427A, // UCHAR
  prn_timeout_4      = 0x427B, // UCHAR
  rs232_timeout_1    = 0x427C, // UCHAR
  rs232_timeout_2    = 0x427D, // UCHAR
  rs232_timeout_3    = 0x427E, // UCHAR
  rs232_timeout_4    = 0x427F, // UCHAR
  keybd_begin        = 0x4180, // UINT
  keybd_end          = 0x4182, // UINT
  video_rows         = 0x4284, // UCHAR
  video_pixels       = 0x4185, // UINT
  video_options      = 0x4287, // UCHAR
  video_switches     = 0x4288, // UCHAR
  video_1_reservd    = 0x4289, // UCHAR
  video_2_reservd    = 0x428A, // UCHAR
  dsk_data_rate      = 0x428B, // UCHAR
  hdsk_status_2      = 0x428C, // UCHAR
  hdsk_error         = 0x428D, // UCHAR
  hdsk_int_flags     = 0x428E, // UCHAR
  hdsk_options       = 0x428F, // UCHAR
  hdsk0_media_st     = 0x4290, // UCHAR
  hdsk1_media_st     = 0x4291, // UCHAR
  hdsk0_start_st     = 0x4292, // UCHAR
  hdsk1_start_st     = 0x4293, // UCHAR
  hdsk0_cylinder     = 0x4294, // UCHAR
  hdsk1_cylinder     = 0x4295, // UCHAR
  keybd_flags_3      = 0x4296, // UCHAR
  keybd_flags_4      = 0x4297, // UCHAR
  timer_wait_off     = 0x4198, // UINT
  timer_wait_seg     = 0x419A, // UINT
  timer_clk_low      = 0x419C, // UINT
  timer_clk_hi       = 0x419E, // UINT
  timer_clk_flag     = 0x42A0, // UCHAR
  lan_1              = 0x42A1, // UCHAR
  lan_2              = 0x42A2, // UCHAR
  lan_3              = 0x42A3, // UCHAR
  lan_4              = 0x42A4, // UCHAR
  lan_5              = 0x42A5, // UCHAR
  lan_6              = 0x42A6, // UCHAR
  lan_7              = 0x42A7, // UCHAR
  video_sav_tbls     = 0x44A8, // ULONG
  days_since_1_80    = 0x41CE, // UINT
  prn_scrn_stat      = 0x5200  // UCHAR
};
```

```
/* BIOS_FP
 * =======
 * creates VOID-far-Pointer to BIOS-Data-area
 * using enum BIOSA
 */
#define BIOSA_FP(l) MK_FP(((l)&0xf000)>>8,(l)&0x00ff)

/* MKB_FP
 * ======
 * creates VOID-far-Pointer to BIOS-Data-area
 * using integer-argument
 */
#define MKB_FP(l) MK_FP(0x40,(l)&0x00ff)

/* BIOS_OFF
 * =======
 * returns Offset of BIOS data
 * using enum BIOSA
 */
#define BIOS_OFF(l) ((l)&0x00ff)

#ifdef __cplusplus

/* class BIOS
 * ==========
 * holds Pointer and Val of BIOS-Data-address
 * usable for all non-Pointer types
 */

template <class T>
class BIOS : public MEM<T>
{
public:
  BIOS(UINT offset)
  : MEM<T>(0x40,offset)
    { Old=Val; }
  BIOS(BIOSA offset)
  : MEM<T>((offset>>8)&0xf0,offset&0x00ff)
    { Old=Val; }
  BIOS(UINT offset,T t)
  : MEM<T>(0x40,offset,t)
    { Old=Val; }
  BIOS(BIOSA offset,T t)
  : MEM<T>((offset>>8)&0xf0,offset&0x00ff,t)
    { Old=Val; }

  virtual T operator () ()
    { Old=Val; return MEM<T>::operator()(); }
  virtual VOID operator = (T t)
    { Old=Val; MEM<T>::operator=(t); }
  BOOL changed()
    { return (Old==Val) ? FALSE : TRUE; }
private:
  T Old;
};

#endif // __MEMABS_HPP
```

Die Klassen MEM und MEM_P werden durch die Programme HC05MM7, HC05MM8, HC05MM9, HC05MM10 getestet. Die Programme HC05MM7, HC05MM8 und HC05MM10 können auch gleichzeitig zum Testen von der später besprochenen Klasse BIOS verwendet werden, die ein Spezialfall der Klasse MEM für den Bereich der BIOS-Daten darstellt.

Anzeige der Adressen der COM-Ports HC05MM7.CPP

Das einfachste Beispiel zeigt, wie man den Inhalt absoluter Speicherstellen ausliest. Es sollen die Adressen der seriellen Schnittstelle angezeigt werden. Man definiert ebensoviele MEM-Objekte als man benötigt,im Beispiel die Schnittstellen COM1..COM4. Die Funktion operator()() liefert den aktuellen Wert der Speicherstelle als Typ UINT.

```
/* HC05MM7.CPP */
/*
 * Anzeige der Adressen der COM-Ports
 * =================================
 */
//#include <membios.h>
#include <memabs.h>

VOID main(VOID)
{
  cout << endl << "Adressen der COM-Ports\n";
//  BIOS<UINT> COM1(rs232_port_1);
  MEM<UINT> COM1(0x0040,0x0000);
  cout << "COM1:" << hex << COM1() << endl;
//  BIOS<UINT> COM2(rs232_port_2);
  MEM<UINT> COM2(0x0040,0x0002);
  cout << "COM2:" << hex << COM2() << endl;
//  BIOS<UINT> COM3(rs232_port_3);
  MEM<UINT> COM3(0x0040,0x0004);
  cout << "COM3:" << hex << COM3() << endl;
//  BIOS<UINT> COM4(rs232_port_4);
  MEM<UINT> COM4(0x0040,0x0006);
  cout << "COM4:" << hex << COM4() << endl;
}
```

Operatorentest HC05MM8.CPP

Bei der Konstruktion von Klassen ist es, genauso wie bei allen anderen Programmprojekten, ratsam, diese durch geeignete Testsituationen einer Bewährungsprobe zu unterziehen und die Testprogramme bei jeder Änderung der Klasse wieder auszuführen und gegebenenfalls zu erweitern.

Im Beispiel HC05MM8.CPP werden alle definierten Operatoren ausgeführt. In den Beispielen wird das MEM-Typ-Objekt a auf die selten benutzte Adresse des seriellen Ports 4 gelegt. In jeder Anweisungsgruppe wird die Adresse mit dem Zuweisungsoperator operator=() initialisiert, der Wert mit der Operator-Funktione operator() zurückgelesen. Danach wird die Operation ausgeführt und das Resultat ausgegeben. Lediglich bei den Operatoren << und >> ergibt sich eine Zweideutigkeit mit den ebenso überladenen Operatoren des Ausgabestreams cout, daher werden die Ausdrücke a<<2 geklammert.

```
/* HC05MM8.CPP */
/*
 * Operatorentest
 * ==============
 */
//#include <membios.h>
#include <memabs.h>

VOID main(VOID)
{
  INT i;
//  BIOS<UINT> a (rs232_port_4);
  MEM<UINT> a (0x0040,0x0006);

  cout << "Using BIOS-Data-Address 40:6 "
          "(=rs232_port_4)\n";
  cout << "for testing Operators\n\n";

  cout << "Testing operators = [abcd] and ()\n";
  a = 0xabcd;
  cout << hex << a() << ' ';
  cout << endl;

  cout << "Testing operators ++,-- starting with 3\n";
  a=3;
  cout << hex << a() << ' ';
  for (i=0; i<4; i++)
    cout << hex << ++a << ' ';
  cout << hex << a() << ' ';
  for (i=0; i<4; i++)
    cout << hex << --a << ' ';
  cout << endl;

  cout << "Testing operators <<,>> starting with 1\n";
  a = 1;
  cout << hex << a() << ' ';
  for (i=0; i<4; i++)
    cout << hex << (a<<2) << ' ';
  cout << hex << a() << ' ';
  for (i=0; i<4; i++)
```

```
    cout << hex << (a>>2) << ' ';
  cout << endl;

  cout << "Testing operator ~ \n";
  a = 0x5aa5;
  cout << hex << a() << ' ';
  ~a;
  cout << hex << a() << ' ';
  cout << endl;

  cout << "Testing operator += [1234+4321] "
                       "and -= [5555-4321]\n";
  a = 0x1234;
  cout << hex << a() << ' ';
  a += 0x4321;
  cout << hex << a() << ' ';
  a = 0x5555;
  cout << hex << a() << ' ';
  a -= 0x4321;
  cout << hex << a() << ' ';
  cout << endl;

  cout << "Testing operator *= [10*123] "
                      "and /= [1230/10]\n";
  a = 0x10;
  cout << hex << a() << ' ';
  a *= 0x123;
  cout << hex << a() << ' ';
  a = 0x1230;
  cout << hex << a() << ' ';
  a /= 0x10;
  cout << hex << a() << ' ';
  cout << endl;

  cout << "Testing operator %= [13%10] "
                       "and &= [1234&00ff]\n";
  a = 0x13;
  cout << hex << a() << ' ';
  a %= 0x10;
  cout << hex << a() << ' ';
  a = 0x1234;
  cout << hex << a() << ' ';
  a &= 0x00ff;
  cout << hex << a() << ' ';
  cout << endl;

  cout << "Testing operator |= [1234|00ff] "
                       "and ^= [1234^00ff]\n";
  a = 0x1234;
  cout << hex << a() << ' ';
  a |= 0x00ff;
  cout << hex << a() << ' ';
  a = 0x1234;
  cout << hex << a() << ' ';
  a ^= 0x00ff;
  cout << hex << a() << ' ';
  cout << endl;

//  BIOS<UINT> *b = new BIOS<UINT>(rs232_port_4);
  MEM<UINT> *b = new MEM<UINT>(0x0040,0x0006);

  cout << "Using BIOS-Data-Address 40:6 "
          "(=rs232_port_4)\n";
  cout << "for testing Operators\n\n";

  cout << "Testing operators = [abcd] and ()\n";
  (*b) = 0xabcd;
  cout << hex << (*b)() << ' ';
  cout << endl;
}
```

Lesen der Uhrzeit aus dem Speicher

Das ursprünglich mit Pointern gelöste Problem wird jetzt mit der Klasse `MEM` angegangen.

```
/* HC05MM3B.CPP */
/*
 * Lesen der Uhrzeit aus den BIOS-Daten, mit MEM-Klasse
 * ================================================
 */
#include <dos.h>
#define P_STREAMS
#include <portable.h>
#include <mytypes.h>
#include <memabs.h>
//#include <membios.h>

VOID main(VOID)
{
  MEM<long> time(0x0040,0x006c);
//  BIOS<long> time(0x6c);
  constream con;
  con.clrscr();
  cout << "Uhrzeit aus dem BIOS-Datenbereich\n";
  do
  {
    cout << time() << ' ';
    delay (1000);
  }
  while (kbhit()==0);
  cin.get();
}
```

Eigene Klasse für Uhrzeit

Hat man oft mit Zeitmessungen zu tun, ist es nützlich die oft wiederkehrenden Umwandlungen zwischen den einzelnen Zeitkomponenten einer eigenen Zeit-Klasse zu übertragen (`HC05MM3C.CPP`).

```
/* HC05MM3C.CPP */
/*
 * Lesen der Uhrzeit aus den BIOS-Daten mit Klasse TIME
 * ================================================
 */
#include <dos.h>
#define P_STREAMS
#include <portable.h>
#include <mytypes.h>
#include <membios.h>

class TIME : public BIOS<long>
{
  long t;
public:
  long operator()(){ t=BIOS<long>::operator()()*0.053; return
t;}
  int h() { return (int)(t/3600); }
  int m() { return (int)(t/60 - h()*60); }
  int s() { return (int)(t - h()*3600 - m()*60); }
  TIME() : BIOS<long>(0x6c) {}
};

VOID main(VOID)
{
  TIME time;

  constream con;
  con.clrscr();
  cout << "Uhrzeit aus dem BIOS-Datenbereich\n";
  do
  {
    time();
    cout << time.h() << ':' << time.m() << ':' << time.s() <<
' ';
    delay (1000);
  }
  while (kbhit()==0);
  cin.get();
}
```

Speicherzugriff über Pointer HC05MM9.CPP

```
#ifndef __TINY__
  #error must use TINY model
#else
```

Dieses Beispiel kompiliert nur im Speichermodell TINY, daher muß dieses Modell in Optionen-Compiler-MemoryModel eingestellt werden.

```
  UINT *mem = new UINT[80];
```

Erzeugt einen Speicherbereich mit 160 Bytes; der Pointer mem zeigt auf den Anfang.

```
  cout << "at " <<hex<< mem <<endl;
  cout << "contents (read by pointers):\n";
```

Der Speicher wird wie folgt belegt, um damit die Memory-Klassen MEM und MEM_P testen zu können.

```
       +-------+
mem    | mem+2 |\
       +-------+ far-Pointer auf Adresse mem+2
mem+1  | DS    |/
       +-------+
mem+2  | 0xab  |
       +-------+
```

```
  *mem = (UINT)(mem+2);
  *(mem+1) = _DS;
  *(mem+2) = 0xab;
```

Diese Speicherbelegung wird zuerst durch Verendung des Pointer mem angezeigt.

Danach werden drei MEM-Objekte off,seg und dat gebildet und die im Speicher befindlichen Daten angezeigt:

```
  MEM<UINT> off(_DS,(UINT)mem);
  MEM<UINT> seg(_DS,(UINT)(mem+1));
  MEM<UINT> dat(_DS,(UINT)(mem+2));
  cout <<hex<< off.off() << ':' <<hex<< off() <<endl;
  cout <<hex<< seg.off() << ':' <<hex<< seg() <<endl;
  cout <<hex<< dat.off() << ':' <<hex<< dat() <<endl;
```

Dasselbe leistet auch ein MEM_P-Objekt m, das auf die Adresse des Pointers initialisiert wird. Das MEM_P-Objekt verwaltet daher zwei Pointer, die auch angezeigt werden: einerseits der in der Ausgabe als 'Primary-Pointer' angegebene, der mit seg() und off() ausgegeben wird, anderseits der Pointer auf den er zeigt, dessen Wert mit pointseg() und pointoff() ausgegeben wird. Die Daten, auf die der Pointer zeigt, werden mit den überladenen Operatoren probehalber manipuliert.

```
  cout << "creating MEM_P object, reading back data\n";
  MEM_P<UINT> m(_DS,(UINT)mem);
  cout << "Primary-Pointer:";
  cout <<hex<< m.seg() << ':'
       <<hex<< m.off() <<endl;
  cout << "points to      :";
  cout <<hex<< m.pointseg() << ':'
       <<hex<< m.pointoff() <<endl;
  cout << "containing data: " <<hex<< m() <<endl;
  cout << "data ++, --, &0x0f, |0xf0" <<endl;
  ++m; cout << m() << ' ';
  --m; cout << m() << ' ';
  m&=0x0f; cout << m() << ' ';
  m|=0xf0; cout << m() << ' ';
  cout <<endl;
```

Dieselben Tests können auch mit einem dynamischen MEM_P-Objekt durchgeführt werden, wie die folgenden Zeilen zeigen:

```
  cout << "creating MEM_P object, reading back data\n";
  MEM_P<UINT> *mp = new MEM_P<UINT> (_DS,(UINT)mem);
  cout << "Primary-Pointer:";
  cout <<hex<< mp->seg() << ':'
       <<hex<< mp->off() <<endl;
  cout << "points to      :";
  cout <<hex<< mp->pointseg() << ':'
       <<hex<< mp->pointoff() <<endl;
  cout << "containing data: " <<hex<< (*mp)() <<endl;
  cout << "data ++, --, &0x0f, |0xf0" <<endl;
  ++(*mp); cout << (*mp)() << ' ';
  --(*mp); cout << (*mp)() << ' ';
  (*mp)&=0x0f; cout << (*mp)() << ' ';
  (*mp)|=0xf0; cout << (*mp)() << ' ';
  cout <<endl;
}
#endif
```

Das MEM_P-Objekt ist bei Behandlung von Datenstrukturen von DOS vorteilhaft anwendbar und wird in einer späteren Folge, die sich mit diesen Zusammenhängen beschäftigt, weiter verwendet.

```
/* HC05MM9.CPP */
/*
 * Pointer-controlled memory access
 * ================================
 */
#ifndef __TINY__
  #error must use TINY model
#else
#include <memabs.h>

VOID main(VOID)
{
  cout << "Pointer controlled memory access\n";
  UINT *mem = new UINT[80];
  cout << "at " <<hex<< mem <<endl;

  cout << "contents (read by pointers):\n";
  *mem = (UINT)(mem+2);
  *(mem+1) = _DS;
  *(mem+2) = 0xab;
  cout <<hex<< mem << ':' <<hex<< *mem <<endl;
  cout <<hex<< (mem+1) << ':' <<hex<< *(mem+1) <<endl;
  cout <<hex<< (mem+2) << ':' <<hex<< *(mem+2) <<endl;

  cout << "contents: (read by MEM-objects)\n";
  MEM<UINT> off(_DS,(UINT)mem);
  MEM<UINT> seg(_DS,(UINT)(mem+1));
  MEM<UINT> dat(_DS,(UINT)(mem+2));
  cout <<hex<< off.off() << ':' <<hex<< off() <<endl;
  cout <<hex<< seg.off() << ':' <<hex<< seg() <<endl;
  cout <<hex<< dat.off() << ':' <<hex<< dat() <<endl;

  cout << "creating MEM_P object, reading back data\n";
  MEM_P<UINT> m(_DS,(UINT)mem);
  cout << "Primary-Pointer:";
  cout <<hex<< m.seg() << ':'
       <<hex<< m.off() <<endl;
  cout << "points to      :";
  cout <<hex<< m.pointseg() << ':'
       <<hex<< m.pointoff() <<endl;
  cout << "containing data: " <<hex<< m() <<endl;
  cout << "data ++, --, &0x0f, |0xf0" <<endl;
  ++m; cout << m() << ' ';
  --m; cout << m() << ' ';
  m&=0x0f; cout << m() << ' ';
  m|=0xf0; cout << m() << ' ';
  cout <<endl;

  cout << "creating MEM_P object, reading back data\n";
  MEM_P<UINT> *mp = new MEM_P<UINT> (_DS,(UINT)mem);
  cout << "Primary-Pointer:";
  cout <<hex<< mp->seg() << ':'
       <<hex<< mp->off() <<endl;
  cout << "points to      :";
  cout <<hex<< mp->pointseg() << ':'
       <<hex<< mp->pointoff() <<endl;
  cout << "containing data: " <<hex<< (*mp)() <<endl;
  cout << "data ++, --, &0x0f, |0xf0" <<endl;
  ++(*mp); cout << (*mp)() << ' ';
  --(*mp); cout << (*mp)() << ' ';
  (*mp)&=0x0f; cout << (*mp)() << ' ';
  (*mp)|=0xf0; cout << (*mp)() << ' ';
  cout <<endl;
}
#endif
```

Beginn und Ende der Tastaturwarteschlange

HC05MM10.CPP

Abschließend werden der Beginn und das Ende der Tastatur-Warteschlange ausgelesen.

MEMBIOS.HPP hat zwei Aufgaben: Einerseits stellt die Klasse BIOS eine Spezialisierung der Klasse MEM dar, da für das Segment der Wert 0x0040 eingesetzt wird. Anderseits erspart die Klasse dem Programmierer, das Wälzen von Tabellen, da der Aufzählungstyp BIOSA ein Verzeichnis aller BIOS-Adressen darstellt. BIOSA hat folgenden Aufbau:

```
typedef enum BIOSA
{
  rs232_port_1   = 0x4100, // UINT
                       ^^...Offset der Variablen
                     ^.....Typ der Variablen
                           1...UINT
                           2...UCHAR
                   ^......Segment (4 oder 5)
  ....
  prn_scrn_stat  = 0x5200  // UCHAR
};
```

Dieser Aufzählungstyp ist nicht nur gemeinsam mit der Klasse BIOS - was nur in C++-Programmen ginge - sondern auch unabhängig davon mit den Makros BIOSA_FP, MKB_FP und BIOS_OFF anwendbar.

BIOS_FP erzeugt einen VOID-far-Pointer und erhält als Argument einen der BIOSA-Elemente. Beispiel:

```
UINT far *Timer_lo = (UINT far *)BIOSA_FP(timer_lo);
```

BIOS_OFF liefert die Offset-Adresse einer BIOSA-Variablen.

```
UINT timer_lo_offset = BIOS_OFF(timer_lo);
```

Die Klasse BIOS wurde gegenüber der Klasse MEM um die Möglichkeit erweitert, ein Objekt mit einem Offset oder mit einer BIOSA-Variablen zu initialisieren. Die Funktion changed() stellt fest, ob sich der Speicherwert zwischen zwei Ablesungen verändert hat.

Die Klasse IBIOS geht davon aus, daß der Inhalt der BIOS-Adresse ein Pointer ist, ähnlich wie bei der Klasse MEM_P. Die Klasse IBIOS wird in erster Linie bei der Handhabung des Tastaturbuffers verwendet.

```
/* HC05MM10.CPP */
/*
 * Beginn und Ende der Tastaturwarteschlange
 * =========================================
 */
//#include <membios.h>
#include <memabs.h>

VOID main(VOID)
{
  cout << endl;
//  BIOS<UINT> begin(keybd_begin);
  MEM<UINT> begin(0x0040,0x0080);
  cout << endl << "Begin of kbd-queue : ";
  cout << hex << begin() << endl;
//  BIOS<UINT> end(keybd_end);
  MEM<UINT> end(0x0040,0x0082);
  cout << endl << "End of kbd-queue : ";
  cout << hex << end() << endl;
}
```

```
/* HC05MM11.CPP */
/*
 * Zugriff auf Interruptvektoren
 * =============================
 */

#include <mytypes.h>
#include <memabs.h>

VOID main(VOID)
{
   MEM<ULONG> harddisktable((UINT)0x0000,(UINT)0x118);
   cout << hex << harddisktable() << endl;
}
```

□