Assembler

Die Königssprache der Programmierer

Wenn Sie die Übungen mit dem Knowhow – Computer erfolgreich durchgestanden haben, dann sind Sie jetzt fit für eine Einführung in die Assemblersprache. Vielleicht gehören Sie zu den Leuten, die sich schon immer scheuten das Thema anzugehen. Assembler? – das ist nur was für Spezialisten.

Sie werden sehen, dass das Programmieren eigentlich viel einfacher als mit einer Hochsprache ist, wenngleich es teilweise umständlicher ist, ein Ergebnis zu erreichen. Ich werde versuchen, Sie ganz behutsam an die Assemblersprache heran zu führen.

Für den, der es gewohnt ist, Assembler zu programmieren, der sollte dieses Kapitel überschlagen. Vieles wird für ihn so selbstverständlich sein, dass es sich nicht lohnt, dieses durchzulesen. Für den Anfänger jedoch können die Erklärungen nicht breit genug ausgedehnt werden. Jede kleine Information ist hier hilfreich.

Ganz am Anfang der Computerei mußte man die Rechner noch in direkter Maschinensprache programmieren. Man mußte wissen, was passiert, wenn man diesen Schalter auf 0 oder 1 stellte. Dieses war natürlich nicht ganz einfach und brachte dem Computer wahrscheinlich den Nimbus „äußerst kompliziert – nur für Spezialisten“ ein.

Interessant dabei ist, dass es am Anfang viele Frauen waren, die diese Spezialität beherrschten.

So gilt Ada Lovelace als erste Programmiererin überhaupt. Sie war die Lebensgefährtin von Charles Babbage, der sein Leben lang versuchte, eine digital arbeitende Rechenmaschine aufzubauen. Ada Lovelace verstand es, die teilweise wirren Gedanken in eine verständliche Sprache umzusetzen.

Grace Hopper verfasste die meisten Programme für den MARK 1 und sie entwickelte nachher - die heute noch benutzte – Sprache COBOL. Nicht zu vergessen: Adele Goldstine schrieb die ersten Programme für die ENIAC – Maschine.

Ein Nachteil des Assemblers ist auch sein Vorteil. Dadurch dass er sehr maschinennah die Programme abarbeitet, nutzt er die Spezialitäten des Prozessors. Bei einem Prozessor anderer Bauart nützt das nichts. Der Assembler A ist für den Prozessor – oder besser meist für die Prozessorfamilie – A geschrieben und läuft nur auf dieser Plattform.

Somit ist es klar, dass wir einen Assembler benötigen, der die Motorolachips 6805 bedienen kann. Ich habe mich an den Freeware – Assembler AS5.EXE gewöhnt, der auf der CD gespeichert ist. So puristisch wie der Assembler sich gibt, so sparsam kann man auch Programme schreiben. Ganze 17 Kilobytes groß ist der AS5.EXE.

Sie werden später sehen, dass die Assemblersprache äußerst effizient arbeitet und die CControl plötzlich zum superschnellen Computer wird. Deshalb ist der Einsatz bei manchen zeitkritischen Problemen angesagt. Oftmals ist der Assembler sogar zu schnell, um mit peripheren Geräten zu kommunizieren. Man muß ihn dann künstlich bremsen.

Außerdem werden Sie sehen, dass die Assemblersprache äußerst platzsparend ist. Nun hat man bei der CControl nicht gerade einen Gigaspeicher zur Verfügung. Gerade einmal 253 Bytes darf das Assemblerprogramm groß sein.

Im Gegensatz zu einer Hochsprache ist der Assembler geradezu Sparweltmeister. Bisher passte alles in diesen 253 Bytes – Rahmen hinein.

Als man Anfang der 50er Jahre den Assembler entwickelte, war dies eine große Vereinfachung für die Programmierer. Ein bestimmter Code bekam ein bestimmtes Symbol, meist aus einer Abkürzung von drei oder 4 Buchstaben bestehend. Bei dem Motorola – Prozessor 6805 sind es 61 Befehle, die zur Verfügung stehen.

Für die meisten Aufgaben werden Sie sicherlich mit einem Befehlssatz von ca 10 Befehlen auskommen. Dieses kann man ganz schnell lernen und letztendlich auch behalten, da die Entwickler wirklich gute Abkürzungen gewählt haben. So geht einem der Befehl **lda** für load akku oder **jsr** für jump subroutine in Fleisch und Blut über.

Bevor wir gleich beginnen, ein kurzer Abriß, wie wir mit dem Assembler die CControl steuern:

Das Programm entwickeln wir in einem normalen Texteditor. Ich benutze dazu den ganz normalen DOS-EDITOR, der mit EDIT Programmname aufgerufen wird. Ist das Programm fertig, so wird es z.B. als TEST.ASM gespeichert. Danach wird das DOS – Programm AS5.EXE TEST.ASM aufgerufen. Dieser Assembler setzt die Programmzeilen in ein Maschinenprogramm um. Aus TEST.ASM wird nun ein TEST.S19.

Diese XYZ.S19 – Programme können dann in die CControl eingelesen werden. Wie gesagt – 253 Bytes – und keines mehr. Das Programm wird im internen EEPROM des Prozessors abgelegt. Die Datenübertragung dauert entschieden länger als die Speicherung der Basicprogramme, die im externen EEPROM 24C65 (8 Beinchen) abgelegt werden. Das Assemblerprogramm wird immer aus Basic heraus aufgerufen mit einem SYS - Befehl.

Den Befehl SYS &H101 sollten Sie sich jetzt schon einmal merken, auch wenn Sie noch nicht wissen, was dies bedeutet. Das Assemblerprogramm wird automatisch in den reservierten Bereich der CControl übertragen, wenn am Ende des Programmes der Befehl SYSCODE , also in unserem Falle SYSCODE „TEST.S19“ steht. Die Datei muß im Verzeichnis des Assemblers liegen.

Ein Tip: Da Sie wahrscheinlich auch von Windows aus nach DOS gehen werden, sollte man sich ein ICON in die Startleiste legen, damit man schnell hin und herschalten kann.

Also: rechte Maustaste auf die Bildschirmoberfläche, NEU – VERKNÜPFUNG wählen. C:\WINDOWS\COMMAND.COM eingeben. Sie können einen Namen vergeben, z.B. Assembler. Und FERTIGSTELLEN.

Danach sollten Sie sich durch ein paar kleine Einstellungen das Leben erleichtern. Wieder rechte Maustaste auf das DOS-Symbol, jetzt Eigenschaften wählen und PROGRAMM auswählen. Unter Arbeitsverzeichnis sollten Sie jetzt den Pfad wählen, in dem sich Ihre CControlprogramme und der Assembler AS5.EXE liegen.

Weitere Hilfe bringt das automatische Laden der Funktion DOSKEY. Jetzt merkt sich DOS die Tastatureingaben – Sie können mit den Cursortasten jetzt einfach zwischen EDIT TEST.ASM und AS5 TEST.ASM hin und herschalten. Dieses sind die beiden Funktionen, die Sie beim Entwickeln ständig benötigen. Das DOS-Symbol sollten Sie dann einfach in die Startleiste ziehen, damit es ständig aufrufbar ist. Ebenso sollten Sie mit dem Editor der CControl CCEW32D.EXE verfahren.

Jetzt können wir endlich loslegen. Ich schlage jetzt vor, dass wir einfach ein fertiges Assemblerprogramm in die CControl einladen, ohne eine genaue Erklärung der Funktion an dieser Stelle zu besitzen. Kopieren Sie von der CD unter dem Verzeichnis ASSEMBLER das Programm LEDON1.S19 in Ihr CControlverzeichnis. Jetzt öffnen Sie den CControleditor und geben in Basic ein:

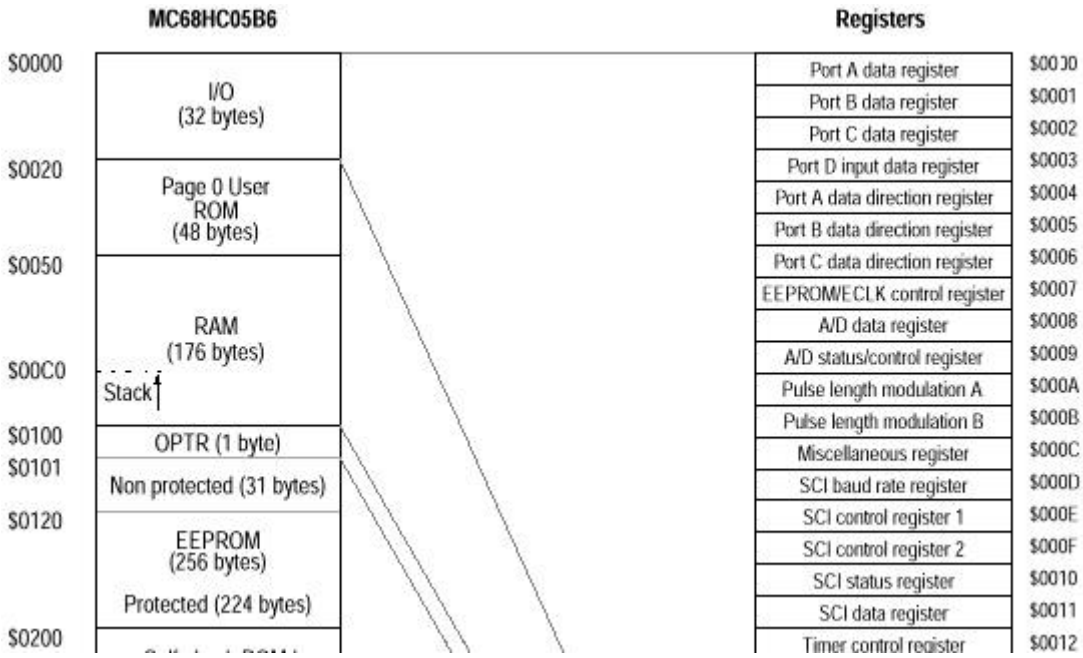
SYS &H0101
SYSCODE „LEDON1.S19“

Das Ganze wird compiliert und in die CControl übertragen. Wenn Sie jetzt das Programm starten, werden Sie etwas Seltsames sehen. Die beiden Leuchtdioden rot (RUN) und gelb (ACTIVE) blinken im Wechsel. Wo steht der Befehl dazu im Basic? Den gibt es nicht.

Dies ist schon der erste Vorteil des Assemblers, den wir hier entdecken können: man kann Dinge bewerkstelligen, die in der Interpretersprache BASIC gar nicht vorgesehen sind. Bei genauer Kenntnis des Prozessors kann man alles ansprechen und ist bei der Programmierung schneller und flexibler.

Jetzt ist doch ein ganzer Berg an Erklärungen zu bewältigen. Ich weiß gar nicht, was jetzt geschickter ist: die Assemblersprache zu erklären oder auf die Spezialitäten des Prozessors einzugehen. Beides benötigen wir zum Verständnis.

Beginnen wir mit dem Prozessor. Ein Ausschnitt aus dem Datenblatt gibt hier Aufschluß (Seite 42 der Dokumentation auf der CD.



Wir sehen hier bei der Aufteilung der Register, der Speicherstellen, dass die ersten 32 Bytes für unsere Arbeit sehr interessant sind. Hier sind die digitalen und analogen Ports zu finden.

Port A liegt auf der Adresse 0, Port B entsprechend auf 1, Port C auf 2. Port D Input data register ist der Analogport. Jeder Digitalport A, B, C hat noch ein sogenanntes Richtungsregister in den Adressen 4, 5, 6.

Mit diesem Wissen kommen wir der Sache mit den blinkenden Leuchtdioden schon näher. Es gibt neben den bekannten Digitalports 1-8 (Port B dataregister) und den Digitalports 9-16 (Port C dataregister) noch einen Port A. Dieser Port wurde vom Entwickler der Software für die CControl für Systemfunktionen reserviert. Hier ist SDA und SCL des I2C-Busses zur Kommunikation mit dem externen EEPROM 24C65 realisiert. Außerdem werden die drei Leuchtdioden grün, gelb und rot geschaltet.

Wir können hier erkennen, wie sparsam man in der Computertechnik arbeiten kann. Das, was uns beim Steuern über die CControl nach aussen viel Spass macht, läuft letztendlich im Prozessor nur in einem Byte ab. Oder besser: in zwei Bytes. Jeder Digitalport hat nämlich noch ein Richtungsbyte. Hier wird bitweise gespeichert, welche Funktion der Port übernehmen soll: Eingang oder Ausgang.

Sitzt das gewählte Bit im Richtungsbyte auf 1, so ist der Port als Ausgang geschaltet, eine 0 macht einen Eingang daraus. Was wir letztendlich wissen müssen: die Aufteilung des Bytes in die einzelnen Bits und deren Funktion.

Dem Schaltplan kann man entnehmen, dass die rote Leuchtdiode auf Bit 2, die gelbe Leuchtdiode auf Bit 3 angeschlossen ist (bei der Zählweise 7 bis 0).

Jetzt können wir zumindest verstehen, wieso die beiden Leuchtdioden auf der CControl blinken können. Das Assemblerprogramm macht es möglich. Wenn wir uns das Programm anschauen, so wird einiges klar werden.

```
*****************************************************************
*
* Assembler      AS5.EXE
* Programmname   LEDON1.ASM
```

```
* COMPILAT      LEDON1.S19, 31 Sysbytes
* Basicprogramm LEDON.BAS
*               SYS &H0101
*               SYSCODE "LEDON.S19"
*
* Demoprogramm zur Verdeutlichung der Benutzung der Assembler-
* prgrammierung fuer den 6805 Motorola - Microcontroller.
* Hier wird gezeigt, dass der Prozessor ueber einen weiteren
* Digitalport verfügt, der von BASIC aus nicht erreichbar ist.
* Der als A-Port bezeichnete 8 Bit breite Digi Ein/Ausgang wird
* bei der CControl fuer Systemaufgaben reserviert. Hier ist SDA und
* SCL des I2C-Busses realisiert. Ausserdem sind die Leuchtdioden
*  ROT, GELB, GRUEN der CControl ueber jeweils einen Bitport
* angeschlossen.
* Mit diesem Assemblerprogramm kann man die beiden Leuchtdioden
* ROT (Bit 2)= RUN und GELB (Bit 3) = ACTIVE benutzen. Es wird
* keine externe Elektronik benötigt.
*  Eigene Versuche zeigten, dass man diese Leuchtdioden benutzen
* kann,
* ohne das Programm zu beeinflussen. Ich hoffe, der Entwickler sieht
* das auch so.
* (c) Wolfgang Back, August 99
*
*****************************************************************


aport     equ  $00      ; in Register 0 ist der Wert des Byteports A
apdir     equ  $04      ; in Register 4 wird die Richtung festgelegt

          org  $101     ; Beginn des Programms

          lda  #255     ; der AKKU wird mit $FF geladen
          sta  apdir    ; die Richtung des Ports: Ausgang

blinken   bclr 2,aport  ; bitclear (ausschalten) von Bit 2 im A-port
          bset 3,aport  ; bitset (einschalten) von Bit 3 im A-port
          jsr  pause    ; jump subroutine = Verzweigung zur Pause
          bset 2,aport  ; bitset (einschalten) von Bit 2 im A-port
          bclr 3,aport  ; bitclear (ausschalten) von Bit 3 im A-port
          jsr  pause    ; jump subroutine = Verzweigung zur Pause
          bra  blinken  ; branch always = Immer zur ‚blinken‘

pause     lda  #255     ; der AKKU wird mit $FF geladen
loop1     ldx  #255     ; das X-Register wird mit $FF geladen
loop2     decx          ; X wird um eins verkleinert - decrementiert
          bne  loop2    ; branch not equal- enn nicht 0, dann loop2
          deca          ; AKKU um eins verkleinern
          bne  loop1    ; branch not equal nach loop1
          rts           ; return to subroutine - hier nach blinken
```

Nehmen wir die beiden ersten Zeilen, die das eben Gesehene widerspiegeln.

```
aport     equ  $00      ; in Register 0 ist der Wert des Byteports A
apdir     equ  $04      ; in Register 4 wird die Richtung festgelegt
```

Wir nennen das Register A **aport**, wir könnten es auch xyz nennen. **equ** bedeutet equal – gleich -.

Und $00 entspricht der Adresse 0, der ersten Registeradresse im Prozessor. Wenn jetzt im Programm irgendetwas mit aport geschieht, so bezieht sich dies auf die Registeradresse 0. **apdir** ist die Bezeichnung für das Richtungsbyte. Wie im Datenblatt zu erkennen ist, gehört hier die Adresse 4 dazu.

```
        org  $101     ; Beginn des Programms
```

Mit **org** legen wir den Programmbeginn fest. An der Adresse $101 (257 dezimal) beginnt das Ablegen des Assemblerprogrammes. Das ist nicht willkürlich gewählt. Blättern Sie einmal zurück zum Datenblatt. Dort sehen Sie, dass bei $100 ein options register reserviert ist. Danach beginnt der freie Userbereich mit 31 Bytes ungeschützt und 224 Bytes geschützt.

Diese Adresse $101 sollten Sie sich merken.

```
         lda  #255     ; der AKKU wird mit $FF geladen
         sta  apdir    ; die Richtung des Ports: Ausgang
```

Eine der wichtigsten Funktionen innerhalb des Assemblers ist das Rechenregister AKKU. Sie werden bei Ihren Assemblerprogrammen oft mit dem AKKU zu tun bekommen. Ein zweites Register kann für viele Funktionen ebenfalls benutzt werden: das X-Register. Auch dieses werden Sie kennenlernen.

Was passiert nun in den obigen Programmzeilen. **lda** = load Akku direkt mit einem Wert, der mit dem Gatter eingeleitet wird. **lda #255** heißt also: lade in das Akkuregister den Wert 255 ($FF).

Die zweite Zeile ist nun besonders wichtig. Denn mit dem Laden des AKKU's ist zunächst einmal nichts erreicht. Erst wenn wir den Befehl **sta apdir** ausführen ist eine wichtige Funktion erfüllt. **Sta** bedeutet store akku, also der Wert des AKKU's wird in das Register **apdir** (Richtungsregister) oder $04 geschrieben. Da wir den AKKU mit 255 geladen haben, so sind im Richtungsbyte alle Bits auf 1 gesetzt. 255 bedeutet: der A - Port ist komplett als Ausgang geschaltet.

Später wird man nicht mehr so brutal einfach alle Bits ein- oder ausschalten. Dann wird man die Bits einzeln betrachten. Z.B. **lda $01 sta apdir** setzt nur das Bit 0 auf 1.

Jetzt wird es schwieriger. Der Hauptteil des Blinkprogrammes sieht so aus:

```
blinken   bclr 2,aport  ; bitclear (ausschalten) von Bit 2 im A-port
          bset 3,aport  ; bitset (einschalten) von Bit 3 im A-port
          bset 2,aport  ; bitset (einschalten) von Bit 2 im A-port
          bclr 3,aport  ; bitclear (ausschalten) von Bit 3 im A-port
          bra  blinken  ; branch always=Immer zur Subroutine blinken
```

Am Anfang steht blinken. Dieses bedeutet: blinken ist eine Sprungadresse eines Unterprogrammes. Innerhalb des Programmes kann man die Subroutine jederzeit anspringen.

Dazu gibt es verschiedene Methoden, die wir später noch kennenlernen. Ein **jsr blinken** bedeutet zum Beispiel jump subroutine blinken. Hier in dieser Routine sehen Sie am Schluß ein **bra blinken** (branch always - verzweige immer). Die Bedeutung ist rechts daneben erklärt.

Dieser Programmteil stellt eine Endlosschleife dar. Das Programm wird von Basic aufgerufen, es geht zur Assemblerroutine und wird dann bis zum Neustart des Prozessors nicht mehr verlassen.

Was passiert?

Die erste Befehlsfolge in der Schleife lautet: **bclr 2 , aport**. Wir wissen noch von oben, dass die rote Leuchtdiode mit Bit 2, die gelbe mit Bit 3 am Digitalport A verbunden ist. **bclr** als bitclear setzt daher das Bit 2 auf 0.

Alle anderen Bits des Ports bleiben so, wie sie waren. Danach wird mit **bset 3,aport** das Bit der gelben Leuchtdiode auf 1 gesetzt, was gleichbedeutend ist mit dem Nichtleuchten der Diode. Bit auf 1 – Leuchtdiode aus; Bit auf 0 – Leuchtdiode an.

Die beiden folgenden Zeilen sind damit auch klar: die Leuchtdioden werden abwechselnd ein- und ausgeschaltet.

Der letzte Befehl **bra blinken** wurde oben schon erklärt. Branch always – also verzweige immer – springt zum Anfang des Unterprogrammes **blinken**.

Wenn wir das Programm in dieser Weise einladen würden, so wäre die Enttäuschung sehr groß. Für unsere Augen würde nichts blinken, für das angeschlossene Oszilloskop sehr wohl. Der Prozessor ist mit dem Assemblerprogramm plötzlich so schnell geworden, dass er im millionstel Sekundentakt die Leuchtdioden ein- und ausschaltet.

Bei der Assemblerprogrammierung weiß man übrigens ganz genau, in welchen Zeitrastern gearbeitet wird. Später wird es manchmal sehr wichtig sein, diese Abarbeitungszeiten genau zu berechnen. Jeder Befehl hat bestimmte Zyklen, die er zum Abarbeiten benötigt. Diese Zyklen kann man im Datenblatt zum 68HC05 auf der CD nachsehen.Der Befehl **bra** z.B. benötigt 3 Zyklen.

Vielleicht ist es an dieser Stelle interessant zu erfahren, mit welchen Taktraten und damit kleinsten Zeiten die CControl arbeitet. Der Quarz auf der CControl taktet mit 4 MHz. Intern werden daraus 2 MHz. 2 MHz bedeuten daher: 1 Taktzyklus beträgt 2 Mikrosekunden.**bra** ist also in 6 Mikrosekunden abgearbeitet.

Wir wollen aber unsere Leuchtdioden blinken sehen. Wir müssen eine Pause nach jedem Vorgang einfügen. Die unten beschriebene Pause besteht aus zwei Schleifen, die nacheinander abgearbeitet werden. Mit einer Schleife wären wir nämlich immer noch zu schnell.

```
pause     lda  #255     ; der AKKU wird mit $FF geladen
loop1     ldx  #255     ; das X-Register wird mit $FF geladen
loop2     decx          ; X wird um eins verkleinert - decrementiert
          bne   loop2     ; branch not equal - wenn nicht null, dann
zu loop2
          deca          ; AKKU um eins verkleinern
          bne  loop1    ; branch not equal nach loop1
          rts           ; return to subroutine-hier nach blinken
```

pause ist hier wieder der Einsprungpunkt des Unterprogrammes. Ein **jsr pause** führt also während des Programmablaufes zur Verzweigung zu diesem Unterprogramm. Eine Rückkehr an die aufrufende Stelle wird hier in der letzten Zeile mit **rts** (return to subroutine) erreicht.

Beim Einsprung wird zunächst das Akkuregister mit dem höchsten Bytewert 255 gefüllt.

Danach ist wieder eine Einsprungmarke mit **loop1** definiert. Das zweite Register, das X-Register wird ebenfalls mit 255 gefüllt. Eine weitere Sprungmarke **loop2** wird definiert.

Ein neuer Befehl (Mnemonic) taucht auf: **decx** und zwei Zeilen später **deca**.
dec ist die Abkürzung von decrement, Verringerung. **decx** bedeutet daher: aus der 255 im X-Register wird eine 254, kommt das Programm nochmals hier vorbei wird eine 253 gespeichert. **deca** ist natürlich auf den Akku bezogen und decrementiert den Inhalt jeweils um 1.

Der Partnerbefehl von **dec** ist **inc**, incrementieren – um 1 erhöhen.

Noch einen neuen Befehl müssen wir erklären:
bne – branch not equal (Verzweige, wenn Inhalt nicht gleich 0 ist)
Der zweite Parameter, der immer dazugehört ist die Verzweigungsmarke. Hier also ein Sprung nach **loop2** im ersten Falle und nach **loop1** in der späteren Zeile.

Was passiert nun?

AKKU und X-Register haben den Inhalt 255.
Der erste Befehl **decx** schreibt 254 in das X-Register. Eine Überprüfung des Inhalts wird mit **bne** durchgeführt. Ist der Inhalt ungleich 0, so wird nach **loop2** verzweigt**. decx** : Inhalt jetzt 253 - Sprung nach **loop2** usw.

Nach 255 Durchläufen ist der Inhalt des X-Registers 0.
Jetzt wird keine Verzweigung mehr durchgeführt, das Programm geht zur nächsten Zeile.

deca heißt es jetzt. Aus dem Akkuinhalt wird eine 254. Das Abprüfen auf 0 über **bne** führt jetzt zu einem Sprung nach **loop1**. Hier wird das X-Register erneut mit 255 gefüllt und das selbe Spielchen beginnt aufs Neue.

Irgendwann ist auch der Inhalt des Akkus 0 und der Rücksprungbefehl **rts** wird erreicht. Die Pause ist zu Ende, das Programm kehrt zum Ende des aufrufenden **jsr** pause im Unterprogramm blinken zurück.

Insgesamt hat der Prozessor also 256 * 256 = 65536 Durchläufe hinter sich. Plötzlich wird die kleine CControl zur Rakete. Ich habe es oben schon angedeutet, dass man jetzt die genaue Pausenlänge sich errechnen kann. Jeder Befehl hat bestimmte Zyklen, die Addition aller Zyklen ergibt die Gesamtpause.

Die oben bei der Erklärung etwas verstummelte Blinkroutine muß jetzt noch mit den Pauseaufrufen ergänzt werden.

```
blinken   bclr 2,aport  ; bitclear (ausschalten) von Bit 2 im A-port
          bset 3,aport  ; bitset (einschalten) von Bit 3 im A-port
          jsr  pause    ; jump subroutine = Verzweigung zur Pause
          bset 2,aport  ; bitset (einschalten) von Bit 2 im A-port
          bclr 3,aport  ; bitclear (ausschalten) von Bit 3 im A-port
          jsr  pause    ; jump subroutine = Verzweigung zur Pause
          bra  blinken  ; branch always=immer zur Subroutine blinken
```

Auf der CD sind noch weitere Programme LEDON gespeichert.
LEDON2.ASM zeigt lediglich, dass man Methoden zur besseren Lesbarkeit des Programmes anwenden sollte.

LEDON3.ASM zeigt die mögliche Wechselwirkung zwischen BASIC und Assembler. Das Programm ist selbsterklärend.

LEDON4.ASM zeigt eine andere mögliche Wechselwirkung zwischen BASIC und Assembler. Das Programm ist selbsterklärend.

Bitmanipulationen mit dem Assembler

Ganz wichtig ist es bei der Arbeit mit der CControl, dass man ein Byte auseinandernehmen kann, dass man einzelne Bits ein- oder ausschalten kann. Auch hierzu gibt es ein Demoprogramm, das ich ausführlich beschreiben will.

Wir benutzen als Anzeigeelemente wieder die beiden Leuchtdioden rot und gelb der CControl. Gezeigt werden soll die Möglichkeit, wie man ein Byte in die entsprechenden Bits zerlegen kann. Die rote Leuchtdiode zeigt hier an, ob ein Bit innerhalb eines Bytes gesetzt ist oder nicht. Die gelbe Leuchtdiode dient hier als optische Taktkontrolle. Auch wenn der Inhalt des zu zerlegenden Bytes 0 ist, so taktet die gelbe Diode 8 mal. So läßt sich das Ergebnis gut kontrollieren.

```
*****************************************************************
**
* Assembler     AS5.EXE
* Programmname  LEDBIT1.ASM
* COMPILAT      LEDBIT1.S19
* Basicprogramm LEDBIT.BAS
*
*               define wert byte
*               #main
*               print "Bitte Wert eingeben"
*               input wert
*               SYS &H0101 wert
*               goto main
*               SYSCODE "LEDBIT1.S19"
* (c) Wolfgang Back, August 99
*
*****************************************************************
*****


aport     equ  $00      ; in Register 0 ist der Wert des Byteports A
apdir     equ  $04      ; in Register 4 wird die Richtung festgelegt

wert      equ  $92      ; Uebergaberegister aus Basic
zaehler   equ  $91      ; Merkervariable

rot       equ  2        ; Bitnummer fuer rote Leuchtdio
gelb      equ  3        ; Bitnummer fuer gelbe Leuchtdiode

          org  $101     ; Beginn des Programms


          bset rot,apdir  ; die Richtung des Bits : Ausgang
          bset gelb,apdir ; die Richtung des Bits : Ausgang

          bset rot,aport  ; rote Leuchtdiode aus
          bset gelb,aport ; gelbe Leuchtdiode aus
```

```
          lda  #8          ; Akku mit 8 laden (8 Bit = 1 Byte)
          sta  zaehler     ; Wert im Register speichern

anfang    lda wert         ; Den Inhalt der Uebergabe aus Basic laden
          lsla             ; schiebt den Inhalt des Akkus nach links
          sta wert         ; den neuen Wert wieder abspeichern
          bcc null         ; branch carry bit clear
          bclr rot,aport   ; Rotbit auf 0
          bra  weiter      ; branch always nach weiter
null      bset rot,aport   ; Rotbit auf 1

weiter    jsr pause        ; springe nach pause
          bclr gelb,aport  ; Gelbbit 0
          jsr pause        ;
          jsr pause        ;
          bset gelb,aport  ; Gelbbit 1
          jsr pause        ;
          jsr pause        ;
          lda zaehler      ; Lade Inhalt von Zaehler
          deca             ; decrementiere Inhalt
          sta zaehler      ; Neuen Wert speichern
          bne anfang       ; Untersuche ob 0, sonst nach Anfang

          bset rot,aport
          bset gelb,aport

          rts            ; springe nach Basic zurueck

pause     lda  #255      ; der AKKU wird mit $FF geladen
loop1     ldx  #255      ; das X-Register wird mit $FF geladen
loop2     decx           ; X wird um eins verkleinert - decrementiert
          bne  loop2     ; branch not equal - wenn <>0, dann zu loop2
          deca           ; AKKU um eins verkleinern
          bne  loop1     ; branch not equal nach loop1
          rts            ; return to subroutine
```

Einige Programmteile sind oben bereits genau erklärt. Hier beschränken wir uns auf den neuen Teil der Bitzerlegung. Ein Byte besteht bekanntlich aus 8 Bits. Deshalb müssen wir auch 8 mal den Bitwert abfragen. Dazu laden wir den Wert 8 in ein am Anfang definiertes Merkerregister zaehler.

```
          lda  #8          ; Akku mit 8 laden (8 Bit = 1 Byte)
          sta  zaehler     ; Wert im Register speichern
```

Da wir 8 mal takten müssen, legen wir eine Sprungmarke mit dem Namen **anfang** an. Von Basic aus wurde ein Byte, das zerlegt werden soll, über **SYS &H0101 wert** mitgeliefert. Der Inhalt kann an der Adresse $92 abgeholt werden. Im Assemblerprogramm habe ich dies auch **wert** genannt. Man könnte es auch ganz anders bezeichnen, da es ja nur auf den Inhalt der Speicherzelle ankommt.

```
anfang    lda wert         ; Den Inhalt der Uebergabe aus Basic laden
```

lsla – ein neuer Befehl. Ausgeschrieben heißt dies logical shift left akku. Der Inhalt des Akkus wird hiermit nach links bitweise rausgeschoben. Der Partnerbefehl zu **lsla** lautet entsprechend **lsra** – logical shift right. Was bedeutet das nun? Der Inhalt von **wert** sei 10101010. Diesen Wert haben wir auch im Akku geladen.

lsla bedeutet nun: das Bit links ist rausgeschoben und steht uns als sogenanntes Carrybit zur Verfügung. Wir sehen das gleich genauer. Der nächste Aufruf von **lsla** liefert uns nunmehr eine 0, da die erste 1 ja bereits rausgeschoben wurde. Und so geht es 8 mal. Damit ist das Byte geleert.

```
          lsla            ; schiebt den Inhalt des Akkus nach links
          sta wert        ; den neuen Wert wieder abspeichern
```

Der neue Wert im Akku wird jeweils im Register **wert** abgespeichert. Der neue Befehl **bcc** bedeutet branch carrybit clear, also verzweige zu einer Sprungmarke, wenn das Carrybit 0 ist, ansonsten gehe weiter im Programm.

Der Partnerbefehl zu **bcc** ist **bcs** (carrybit set). Hier könnten wir beide Befehle benutzen. Wir fragen das geshiftete Carrybit ab. Ist es 0, so springen wir zur Sprungmarke **null**. Ist das Carrybit 1, löschen wir die Rotdiode mit **bclr rot,aport**. Danach verzweigen wir mit dem bereits erklärten Befehl branch always zur Sprungmarke weiter. Ist das Carrybit 0, so landen wir bei der Sprungmarke **null**. Hier wird die Diode gelöscht, das Bit wird gesetzt.

```
          bcc null        ; branch if carrybit clear
          bclr rot,aport  ; Rotbit auf 0
          bra  weiter     ; branch always nach weiter
null      bset rot,aport  ; Rotbit auf 1
```

Der Rest ist einfach zu verstehen. Hier werden nur ein paar Pausen gesetzt, damit man das Shiften gut beobachten kann. Außerdem wird die gelbe Leuchtdiode ein- und ausgeschaltet.

```
weiter    jsr pause       ; springe nach pause
          bclr gelb,aport ; Gelbbit 0
          jsr pause       ;
          jsr pause       ;
          bset gelb,aport ; Gelbbit 1
          jsr pause       ;
          jsr pause       ;
```

Hier muß noch etwas erklärt werden.Wir wollen 8 Takte erzeugen, um das Byte komplett auszulesen. Dazu nutzen wir ein Hilfsregister mit Namen **zaehler**.

Vor diesem Unterprogramm haben wir bereits in das Register eine 8 geladen.
Den Inhalt rufen wir mit **lda zaehler** wieder in den Akku. Danach kommt der bekannte Befehl **deca** (decrementiere akku). Aus der geladenen 8 wird eine 7. Dieser Wert wird wieder in den **zaehler** zurückgeschrieben. Danach 6,5,4,3,2,1,0

```
          lda zaehler     ; Lade Inhalt von Zaehler
          deca            ; decrementiere Inhalt
          sta zaehler     ; Neuen Wert speichern
```

Auch hier dieser Befehl **bne** (branch not equal). Der Akku wird auf den Wert 0 getestet. Ist die 0 erreicht, so geht das Programm weiter, ansonsten springt es zum Anfang. Danach werden beide Leuchtdioden ausgeschaltet, um das Ende der Aktion zu signalisieren. Mit dem Befehl **rts** (return to subroutine) geht das Assemblerprogramm zurück in das aufrufende Basic.

```
          bne anfang      ; Untersuche ob 0, sonst nach Anfang

          bset rot,aport
```

```
          bset gelb,aport

          rts              ; springe nach Basic zurueck
```

Um dieses Programm bedienen zu können, sollten Sie ein Terminalprogramm geladen haben. Entweder Hyperterminal oder mein speziell für die Arbeit mit der CControl entwickeltes Teminalprogramm TERMINAL.EXE auf der CD.

Hier können Sie dann verschiedene Bytewerte übergeben und das Ergebnis an den Leuchtdioden ablesen.

Hiermit haben wir sogar ein technisches Prinzip realisiert: das serielle Übertragen eines Bytes unter Zuhilfenahme eines Taktsignals. In vielen technischen Systemen wird so gearbeitet.

Eine primitive Ampel

Ein weiteres Programm soll nun verdeutlichen, wie man innerhalb eines Assemblerprogrammes auf Eingaben von aussen reagieren kann. Wieder sind es die Leuchtdioden, die das Ergebnis anzeigen. Diesmal wird die grüne Leuchtdiode mitbenutzt, obwohl sie sich nur bedingt zum Leuchten bringen lässt. Da diese Diode eine angelegte Frequenz anzeigt, wird sie vom Programm ständig geschaltet. Zum Aufblitzen reicht es aber.

Rot, gelb, grün – das klingt nach einer Ampel. Dieses wollen wir jetzt benutzen, um durch Eingaben aus dem Basic zu verschiedenen Reaktionen zu kommen. Mitunter habe ich das Gefühl, dass die wirklichen Ampeln in Köln in ihrer Grünphase ähnlich geschaltet sind – Sie werden sehen, warum.

Dazu führen wir den Befehl **cmp** compare (vergleiche) ein. Das Basicprogramm von oben kann weitergenutzt werden, wenn die neue Datei ampel.s19 im syscode eingefügt wird. Außerdem kommt der Befehl **beq** branch if equal (verzweige, wenn gleich) hinzu.

```
*****************************************************************
**
* Assembler     AS5.EXE
* Programmname  AMPEL.ASM
* COMPILAT      AMPEL.S19
* Basicprogramm AMPEL.BAS
*
*               define wert byte
*               #main
*               print "Bitte Wert eingeben (1-3)"
*               input wert
*               SYS &H0101 wert
*               goto main
*               SYSCODE "AMPEL.S19"
* (c) Wolfgang Back, August 99
*****************************************************************


aport     equ  $00      ; in Register 0 ist der Wert des Byteports A
apdir     equ  $04      ; in Register 4 wird die Richtung festgelegt
wert      equ  $92      ; Uebergaberegister aus Basic
```

```
rot       equ  2        ; Bitnummer fuer rote Leuchtdiode
gelb      equ  3        ; Bitnummer fuer gelbe Leuchtdiode
gruen     equ  4        ; Bitnummer fuer gruene Leuchtdiode

          org  $101     ; Beginn des Programms


          bset rot,apdir  ; die Richtung des Bits : Ausgang
          bset gelb,apdir ; die Richtung des Bits : Ausgang
          bset gruen,apdir; die Richtung des Bits : Ausgang

          bset rot,aport  ; rote Leuchtdiode aus
          bset gelb,aport ; gelbe Leuchtdiode aus
          bset gruen,aport; gruene Leuchtdiode aus

          lda  wert       ; Akku mit Uebergabebyte laden
          cmp #1
          beq rotein
          cmp #2
          beq gelbein
          cmp #3
          beq gruenein
          rts

rotein    bclr rot,aport
          rts
gelbein   bclr gelb,aport
          rts
gruenein  bclr gruen,aport
          rts
```

Vieles ist beim Alten geblieben und braucht nicht erklärt zu werden. Neu ist der untere Teil:

```
          lda  wert       ; Akku mit Uebergabebyte laden
          cmp #1
          beq rotein
          cmp #2
          beq gelbein
          cmp #3
          beq gruenein
          rts
```

Der Akku wird geladen mit dem Registerwert `wert`, der von Basic mitgegeben wurde. **Cmp #1** vergleicht nun den Akkuinhalt mit dem Wert 1. Steht im Akku eine 1, ist der Vergleich erfolgreich, so wird über **beq** eine Verzweigung zur Sprungmarke **rotein** vorgenommen. Die Leuchtdiode wird eingeschaltet und der nächste Befehl **rts** springt zurück nach Basic. Damit sind wohl auch die nächsten Funktionen mit gelb und grün erklärt.

Multiplikation

Ich versuche bei unserer primitiven Anzeigemethode zu bleiben, um den Befehl **mul** multiplicate (multipliziere) zu erklären. Nicht jeder Assembler hat diesen Befehl. Für den Anfänger ist er etwas gewöhnungsbedürftig, weil man das Ergebnis (wenn es größer als 255

ist) aus zwei Registern holen muß. Diese Methode muß man jedoch beherrschen, wenn man später mit 16 Bit breiten Werten (word) zu tun hat.

Bei der Multiplikation wird der Wert, des Akkus mit dem Wert des X-Registers multipliziert. Das Ergebnis steht dann als Highbyte im X-Register und als Lowbyte im Akku.

Beispiel 1: 2 * 4 ergibt 8.
Also Highbyte 0, Lowbyte 8.

Beispiel 2: 2 * 128 ergibt 256
Highbyte 1, Lowbyte 0

Bei größeren Werten ist die Zahl aus dem Highbyte mit 256 zu multiplizieren, die Zahl aus dem Lowbyte dann zu addieren.

Auch wenn dieses Programm keinen allzugrossen praktischen Nutzen hat, so kann man daran doch Grundsätzliches üben. Letztendlich zeigt es die Multiplikationsergebnisse sogar an. Die gelbe Leuchtdiode taktet den Inhalt des Highbytes durch Blinken raus, die rote Leuchtdiode übernimmt den Part für das Lowbyte

Viele Routinen sind auch hier bereits bekannt und müssen nicht extra erklärt werden.

```
*****************************************************************
**
* Assembler     AS5.EXE
* Programmname  MULTI.ASM
* COMPILAT      MULTI.S19
* Basicprogramm MULTI.BAS
*
*               define wert1 byte
*               define wert2 byte
*               #main
*               print "Bitte Wert 1 eingeben (1-3)"
*               input wert1
*               print "Bitte Wert 2 eingeben (1-3)"
*               input wert2
*               SYS &H0101 wert1,wert2
*               goto main
*               SYSCODE "MULTI.S19"
* (c) Wolfgang Back, August 99
*
*****************************************************************

aport     equ  $00      ; in Register 0 ist der Wert des Byteports A
apdir     equ  $04      ; in Register 4 wird die Richtung festgelegt

wert1     equ  $94      ; Uebergaberegister aus Basic
wert2     equ  $92      ; Uebergaberegister aus Basic

rot       equ  2        ; Bitnummer fuer rote Leuchtdiode
gelb      equ  3        ; Bitnummer fuer gelbe Leuchtdiode
          org  $101       ; Beginn des Programms

          bset rot,apdir  ; die Richtung des Bits : Ausgang
          bset gelb,apdir ; die Richtung des Bits : Ausgang
          bset rot,aport  ; rote Leuchtdiode aus
          bset gelb,aport ; gelbe Leuchtdiode aus
```

```
          lda  wert1       ; Akku mit Uebergabebyte1 laden
          ldx  wert2       ; Akku mit Uebergabebyte2 laden

          mul              ; Multipliziert Akku mit X-Register

          sta  wert1       ; das Lowergebnis speichern
          stx  wert2       ; das Highergebnis speichern

          jsr  highwert    ; Highbyte anzeigen
          jsr  lowwert     ; Lowbyte anzeigen

zurueck   rts              ; zurueck nach BASIC

lowwert   lda wert1        ; gespeicherten Lowwert laden
          cmp #0           ; ist er 0
          beq zurueck      ; dann springe zurueck, Ende

decakku   lda wert1        ; lade Lowwert
          deca
          sta wert1        ; 1 abziehen
          bclr rot,aport   ; rote Leuchtdiode an
          jsr pause
          jsr pause
          bset rot,aport   ; rote Leuchtdiode aus
          jsr pause
          jsr pause
          lda wert1        ; lade den neuen Lowwert
          bne decakku      ; vergleiche auf 0
          rts              ; Ruecksprung


highwert  lda wert2        ; lade Highwert
          cmp #0           ; vergleiche auf 0
          beq zurueck      ; verzweige zurueck, BASIC
decxreg   lda wert2        ; wie oben
          deca
          sta wert2
          bclr gelb,aport
          jsr pause
          jsr pause
          bset gelb,aport
          jsr pause
          jsr pause        ; Wert2 muss neu geladen werden, da
          lda wert2        ; der Akku in pause veraendert wurde
          bne decxreg      ; vergleiche auf 0
          rts


pause     lda  #255     ; der AKKU wird mit $FF geladen
loop1     ldx  #255     ; das X-Register wird mit $FF geladen
loop2     decx          ; X 1 weniger - decrementiert
          bne  loop2    ; branch not equal- wenn nicht 0,dann loop2
          deca          ; AKKU um eins verkleinern
          bne  loop1    ; branch not equal nach loop1
          rts           ; return to subroutine
```

Ich hoffe, dass das Programm ordentlich startet. Sie werden aufgefordert, zwei Werte einzugeben. Diese Werte werden multipliziert. Probieren Sie mehrere Wertepaare aus und beobachten Sie dabei die Inhalte des Hi- und des Lowbytes.

Eine solche Uhr hat niemand.

Die ultimative serielle Assemblerclock

Ich kann es nicht lassen. Nachdem ich diese drei Leuchtdioden entdeckt habe, fällt mir immer wieder eine Anwendung ein. Mit dieser aber soll es dann gut sein. Aber ich hoffe, dass Sie einen Einstieg in die Assemblersprache erhalten haben. Mit dieser ultimativen Uhr können Sie sich die Zeit jeweils beim Minutenwechsel seriell ausgeben lassen. Ganz ernst gemeint ist die Anwendung nicht, doch wer hat schon so etwas? Das Programm ist leicht zu verstehen, da es dem Multiplikationsprogramm ähnelt. Die Stunden und Minuten werden wie vorhin aus Basic dem Assembler übergeben. Viel Spaß mit Ihrer neuen Uhr. Hoffentlich kommen Sie beim Ablesen der Zeit nicht durcheinander. Gelb sind die Stunden, rot die Minuten.

```
*****************************************************************
**
* Assembler     AS5.EXE
* Programmname  SERIUHR.ASM
* COMPILAT      SERIUHR.S19
* Basicprogramm SERIUHR.BAS
*              define altzeit byte
*               #main
*               if rxd then get altzeit
*               if minute <> altzeit then SYS &H0101 hour,minute
*               altzeit = minute
*               goto main
*               SYSCODE "SERIUHR.S19"
* (c) Wolfgang Back, August 99
*
*****************************************************************

aport     equ  $00      ; in Register 0 ist der Wert des Byteports A
apdir     equ  $04      ; in Register 4 wird die Richtung festgelegt

wert1     equ  $94      ; Uebergaberegister aus Basic
wert2     equ  $92      ; Uebergaberegister aus Basic


rot       equ  2        ; Bitnummer fuer rote Leuchtdiode
gelb      equ  3        ; Bitnummer fuer gelbe Leuchtdiode

          org  $101     ; Beginn des Programms


          bset rot,apdir  ; die Richtung des Bits : Ausgang
          bset gelb,apdir ; die Richtung des Bits : Ausgang

          bset rot,aport  ; rote Leuchtdiode aus
          bset gelb,aport ; gelbe Leuchtdiode aus


          jsr  stunden    ; Highbyte anzeigen
```

```
          jsr  minuten     ; Lowbyte anzeigen

zurueck   rts              ; zurueck nach BASIC

stunden   lda wert2        ; lade Stunden
          cmp #0           ; vergleiche auf 0
          beq zurueck      ; verzweige zurueck, BASIC
decxreg   lda wert2        ; wie oben
          deca
          sta wert2
          bclr gelb,aport  ; gelb an
          jsr pause
          jsr pause
          bset gelb,aport  ; gelb aus
          jsr pause
          jsr pause        ; Wert2 muss neu geladen werden, da
          lda wert2        ; der Akku in pause veraendert wurde
          bne decxreg      ; vergleiche auf 0
          rts

minuten   lda wert1        ; gespeicherten Minutenwert laden
          cmp #0           ; ist er 0
          beq zurueck      ; dann nsch Basic zurueck, Ende

decakku   lda wert1        ; lade Lowwert
          deca
          sta wert1        ; 1 abziehen
          bclr rot,aport   ; rote Leuchtdiode an
          jsr pause
          jsr pause
          bset rot,aport   ; rote Leuchtdiode aus
          jsr pause
          jsr pause
          lda wert1        ; lade den neuen Minutenwert
          bne decakku      ; vergleiche auf 0
          rts              ; Ruecksprung

pause     lda  #255     ; der AKKU wird mit $FF geladen
loop1     ldx  #255     ; das X-Register wird mit $FF geladen
loop2     decx          ; X wird um eins verkleinert - decrementiert
          bne  loop2    ; branch not equal- wenn <>0, dann zu loop2
          deca          ; AKKU um eins verkleinern
          bne  loop1    ; branch not equal nach loop1
          rts           ; return to subroutine
```

Die serielle Schnittstelle mit Assembler bedienen

Es ist doch noch nicht zu Ende mit unseren beiden Leuchtdioden. Sie sollen auch bei dem nächsten Programm zur Anzeige benutzt werden. Es soll nun die serielle Schnittstelle benutzt werden, um gezielt die gelbe oder rote Leuchtdiode einzuschalten oder auszuschalten. Die serielle Schnittstelle verfügt über 4 Register, die im Datenblatt aufgeführt sind. Ein Ausschnitt verdeutlicht dies.

SCI baud rate register	$000D
SCI control register 1	$000E
SCI control register 2	$000F
SCI status register	$0010
SCI data register	$0011

$000D repräsentiert das Register, in dem die gewünschte Baudzahl zur Datenübertragung festgelegt wird. Die höchste erreichbare serielle Geschwindigkeit bei einem Quarz von 4 MHz beträgt hier 9600 Baud. Mit dieser Geschwindigkeit arbeitet die CControl. Es ist ziemlich kompliziert, die einzustellenden Werte genau zu verstehen, da man hier mit Teilerverhältnissen arbeiten muss, die durch die einzelnen Bits des Baudregisters eingestellt werden. Wir geben hier den Wert $C0 (192) ein, der eine Baudrate von 9600 Bits/sec einstellt.

In $000E befindet sich das erste Controlregister der seriellen Schnittstelle. Hier kann man die Art der Datenübertragung einstellen: synchron oder asynchron. Da wir mit der normalen seriellen Schnittstelle asynchron arbeiten, erhält dieses Register den Wert $00.

In $000F befindet sich das zweite Controlregister der seriellen Schnittstelle. Hier kann man Interrupts ein- und ausschalten. Ebenso wird der Empfänger ein- und ausgeschaltet. Interrupts lassen wir ausser Betracht und füllen das Register mit Sender **ein** und Empfänger **ein** (Bit 3 und 2) = 13 oder $0C.

In $0010 ist das Statusregister zur seriellen Schnittstelle angesiedelt. Hier kann man ständig abfragen, ob sich auf der seriellen Leitung etwas getan hat. Bit 5 zeigt an, dass ein neues Datenbyte geschickt wurde. Bit 7 signalisiert: die Schnittstelle ist frei – es kann ein Byte gesendet werden.

In $0011 befindet sich das Datenregister. Hier werden die empfangenen und gesendeten Werte abgelegt.

Mit diesem Wissen können wir jetzt ein einfaches Assemblerprogramm realisieren. Sie sollten sich dieses Programm gut merken, denn es kann später als Monitorprogramm innerhalb des Assemblers benutzt werden: Falls man nicht genau weiss, wo der Assembler landet, kann man sich hierüber Ausgaben generieren.

Ist das Terminalprogramm gestartet, so kann man mit der Eingabe einer 1 oder einer 2 die Leuchtdioden wechselweise einschalten und ausschalten. Gleichzeitig wird zur Demonstration der umgekehrte Weg gegangen. Das eingehende Signal wird über **writecom** wieder zurück auf die Schnittstelle geschrieben.

Hier lernen wir zwei weitere wichtige Möglichkeiten der Bitmanipultion kennen. **Brclr** und **brset.** Auch dies sind Branchbefehle, Verzweigungsbefehle, die effektiv eingesetzt werden können. **brclr** heißt: branch if bit is clear, entsprechend **bset** = branch if bit is set. Bei writecom und readcom sind diese Befehle eingesetzt.

Untersuchen wir das Beispiel an dem Unterprogramm **readcom**:

```
readcom   brclr 5,status,readcom ; warte auf bit 5 =1
          lda daten
          rts
```

Beim Einsprung aus dem Programm zu **readcom** trifft es auf den Branchbefehl **brclr**. Abgefragt wird das Bit 5 im Statusregister **status**. Die Verzweigung geht wieder zu **readcom**.

Im Klartext heißt dies: Verzweige so lange in einer Schleife, bis das Bit 5 nicht mehr 0 ist. Mit **brset** wird auf Bit 0 abgefragt. Diese beiden Befehle kann man sehr bequem zum Einsatz bringen.

```
*****************************************************
* Demoprogramm zur Nutzung der serielle Schnittstelle
* unter Assembler. Die Schnittstelle wird auf die Bytes
* 49 (1) und 50 (2) abgefragt.
* Die beiden Leuchtdioden auf der CControl werden ein-
* und ausgeschaltet.
* Programmname SER1.ASM
* Compilat:    SER1.S19
* Basic:       SYS &H0101
*              syscode "ser1.s19"
* (c) Wolfgang Back, August 1999
*****************************************************

aport     equ $00 ; A-Port
apdir     equ $04 ; A-Port-Richtung
rot       equ 2   ; Bit 2
gelb      equ 3   ; Bit 3

baudrate  equ $0D ; Baudratenregister
cr1       equ $0E ; Controlregister 1
cr2       equ $0F ; Controlregister 2
status    equ $10 ; Statusregister
daten     equ $11 ; Datenregister

          org $101

          bset gelb,apdir; Bit 3 als Ausgang
          bset rot,apdir ; Bit 2 als Ausgang


seriell   lda #$C0       ; 9600 Baud
          sta baudrate
          lda #$00       ; keine Interrupts
          sta cr1
          lda #$0C       ; Sender und Empfaenger ein
          sta cr2

main      jsr readcom    ; gehe nach read
          cmp #49        ; vergleiche Akkuinhalt
          beq rotan      ; 49? dann rotan
          cmp #50        ; vergleiche Akkuinhalt
          beq gelban     ; 50? dann gelban
          bra main       ; branch always nach main

rotan     bset rot,aport ; rot ein
          bclr gelb,aport; gelb aus
          jsr writecom
```

```
          jsr main

gelban    bset gelb,aport; gelb ein
          bclr rot,aport ; rot aus
          jsr writecom
          jsr main

readcom   brclr 5,status,readcom ; warte auf bit 5 =1
          lda daten
          rts

writecom  brclr 7,status,writecom; Schnittstelle frei?
          sta daten
           rts
```

Vielleicht schauen wir uns noch einen Befehl an, der manchmal recht nützlich sein kann. Mit dem Befehl **com** kann man ein Byte komplementär (invertieren) darstellen. Aus dem Bitmuster 00000000 wird dann 11111111, oder aus 65 = 01000001 wird entsprechend 10111110.

Sowohl das Basic Programm, wie das Assemblerprogramm, sind so einfach, dass sie nicht abgespeichert wurden. Mit dem Kopierbefehl kann man das kleine Programm in den Ccontroleditor bringen. Das Assemblerprogramm kann man sicher mit der Hand eingeben.

Basicprogramm
:

```
define ausgabe byte
define wert byte
define i byte

#main
input wert
print
for i=7 to 0 step -1
if wert and 1 shl i then print „1 „; else print „0 „;
next i
sys &H0101 wert
print
for i=7 to 0 step -1
if ausgabe and 1 shl i then print „1 „; else print „0 „;
next i
goto main
syscode „complem.s19“
```

Assemblerprogramm:

```
ausgabe equ  $A1
wert    equ  $92

        org $101
        lda  wert
        coma
        sta  ausgabe
        rts
```

Wie in Basic, so lassen sich im Assembler auch Tabellen anlegen. Manchmal kann dies eine bequeme Methode sein, um sich Werte oder Bitmuster zu merken. Bei dem AS5 – Assembler ist dies auf folgende Weise zu realisieren.

Die Tabelle wird am Ende des Programmes angelegt. Sie erhält einen Namen, der während des Programmablaufs angesprochen wird. Die Tabellenzeilen werden mit **fcb** eingeleitet. Ein Beispiel:

```
tabelle
fcb 1,2,3,4,5,6
fcb 7,8,9,10,11
```

Die einzelnen Ziffern werden im Programm indiziert gelesen. Der Index wird in das X-Register eingelesen. Im Akku wird dann der entsprechende Tabellenwert gespeichert.

```
            Ldx #0
            Lda tabelle,x
```

Im Akku steht nach dieser Operation eine 1, da der Indexwert 0 der erste Tabelleneintrag ist.

Natürlich kann man auch direkt indizieren: `lda tabelle,3` holt den 4. Wert aus der Tabelle (eine 5).

Zum Schluß unserer Hauptbefehlsreihe wollen wir noch den Befehl **nop** betrachten. So unsinnig dieser Befehl bei der ersten Betrachtung aussehen mag, so wichtig kann er sein, wenn man zeitkritische Programme schreibt. **nop** bedeutet ganz einfach no operation. Dieser Befehl im Programm verändert den Ablauf nur in soweit, dass er zur Abarbeitung 2 Zyklen benötigt. Daher kann er günstig eingesetzt werden, wenn man Pausen kurzer Länge aufbauen möchte. Auch wenn ein Programm unsymmetrisch in der Abarbeitung ist, so kann man mit **nop**s die andere Hälfte angleichen.

Wir haben jetzt schon einige Befehle des Assemblers kennengelernt. Insgesamt sind es 61, die bei der Motorola CPU existieren. Viele davon wird man nie benötigen. Zählen wir einfach einmal auf, was wir geschafft haben. Sehen Sie sich die Befehle an und denken Sie darüber nach, ob Sie deren Funktion behalten haben.

lda, ldx, deca, decx, sta, stx, bset, bclr, brclr, brset, bcc, bcs, bne, beq, cmp, bra, rts, jsr, mul, org, com, nop, inc, dec

Nach diesen Übungen wollen wir uns einmal anschauen, wie wir zeitkritische Programmteile bei Lallus vom Assembler erledigen lassen können. Das lahmste Pferd im Basicprogramm ist der I2C-Bus.

Da hier viele Takte ausgeführt werden müssen, damit eine Information gelesen oder geschrieben wird, macht es Sinn, diese Routine auszulagern. Es ist gar nicht so schwer den Ablauf zu verstehen. Alle Befehle, die dazu benötigt werden, wurden bereits besprochen. Der Assembler dürfte hier sicherlich um ein Vielfaches schneller sein als die Interpretersprache Basic. Schauen wir uns die komplette Schreib / Leseroutine an.

```
*****************************************************
*         I2C - Schreib/Leseroutine                 *
*      Es wird die Adresse fuer den PCF-Chip,       *
* der Wert und die Richtung aus Basic uebergeben.   *
*       Beim Lesevorgang übergibt das               *
*      Register -Ausgabe- den Wert an Basic         *
```

```
*       © Wolfgang Back, Pfingsten 1999              *
* Programmname: I2CASM.ASM                           *
* Compilat:     I2CASM.S19                           *
* Basic:        SYS &H0101 adresse,wert,richtung     *
*               SYSCODE „I2CASM.S19“                 *
*****************************************************
bport    equ $01              ; Digitalports 1 - 8
bpdir    equ $05              ; Richtung port B

data     equ 7                ; Dataport (Port 8)
clock    equ 6                ; Clockport(Port 7)

richtung equ $92              ; Schreiben(1),Lesen(0)
i2cwert  equ $94              ; Stackwert (Uebergabe
adresse  equ $96              ; aus Basic)

ausgabe  equ $A1              ; Basic Useradresse

merker   equ $91              ; RAM - Variable
wert     equ $93              ; RAM - Variable
*****************************************************
         org   $101           ; Startadresse

         lda   #192           ;
         sta   bpdir          ; B-Port (7 und 6 als Ausgang)

start    bset  clock,bport    ; clock 1 setzen
         bset  data,bport     ; data  1 setzen
         bclr  data,bport     ; data  0 setzen
         bclr  clock,bport    ; clock 0 setzen

main     lda   adresse        ; Adresswert laden
         jsr   i2cwrite       ; Adresse schreiben
         jsr   getack         ; warten auf acknowledge
         lda   richtung       ; Lesen oder schreiben?
         bne   write          ; wenn nicht 0, dann write
         jsr   i2cread        ; gehe zur Leseroutine
         jsr   stop

write    lda   i2cwert
         jsr   i2cwrite
         jsr   getack

stop     bclr  data,bport     ; data  0 setzen
         bset  clock,bport    ; clock 1 setzen
         bset  data,bport     ; data  1 setzen

ende     rts                  ; zurueck nach Basic
         end

i2cwrite ldx   #8             ; Shiftzaehler laden

shift    lsla                 ; logical shift left
         bcc   null           ; verzweige wenn carrybit 0
         bset  data,bport     ; data auf 1
         bra   eins           ; verzweige nach weiter
null     bclr  data,bport     ; data auf 0
```

```
eins     jsr   pulse           ; pulsen
         decx                  ; Shiftzaehler -1
         bne   shift           ; verzweige nach shift
         rts

i2cread  ldx   #8              ; Schleifenzaehler laden
         lda   #128            ; Wertigkeit laden
         sta   wert            ; speichern in wert
         bclr  data,bpdir      ; data auf Eingang stellen

loop     brclr data,bport,zero ; verzweige, wenn data 0
         lda   merker          ; lade merker
         add   wert            ; addiere Shiftergenis
         sta   merker          ; speichern
zero     lda   wert            ; lade Shiftvariable
         lsra                  ; logisch rechts shiften
         sta   wert            ; speichern
         jsr   pulse           ; pulsen
         decx                  ; Schleife um 1 verkleinern
         bne   loop            ; verzeige nach loop
         lda   merker          ; lade Merkervariable
         sta   ausgabe         ; Ausgabe an Basic
         rts

pulse    bset  clock,bport     ; clock 1 setzen
         bclr  clock,bport     ; clock 0 setzen
         rts

getack   bset  data,bport       ; data 1 setzen
warte    brclr data,bport,warte ; auf data 0 warten
         jsr   pulse            ; pulsen
         rts
```

Wenn man das Programm genauer studiert und in Basic bereits den I2C-Bus realisiert hat, so kommen einem die Programmteile sicherlich bekannt vor. Da das Programm gut dokumentiert ist, verzichte ich hier auf eine genaue Erklärung der einzelnen Programmabschnitte. Alle Befehle, die hier im Assembler vorkommen, wurden vorher bereits besprochen.

Der Vorteil der Assemblersprache kann hier am I2C-Bus auch in einem anderen Zusammenhang gesehen werden. Beim Lallusprojekt haben wir die beiden seriellen Leitungen des I2C-Busses auf Port 8 und Port 7 gelegt.

Dieses ist eigentlich eine Verschwendung der Digitalports, da SDA und SCL bei der CControl bereits an eigene Pins rausgeführt wurden. Diese Ports können jedoch von Basic aus nicht angesprochen werden. Daher gingen wir den oben beschriebenen Weg. Dies hätte nämlich bedeutet, dass man sich von Anbeginn mit dem Assembler auseinandersetzen müßte Ansonsten hätte man die I2C-Funktionen nur über den Assembler erreichen können. Eigentlich schade.

Wie wir an den Programmen LEDON gesehen haben, ist mit der Assemblersprache jedes Register erreichbar. Die bei der CControl herausgeführten Ports SDA und SCL sind mit dem A – Register verbunden (Bit 0 und Bit 1). Wie oben gesehen, kann man dieses Register ansprechen. Damit kann man bei eigenen Anwendungen SDA und SCL des Systems benutzen.

Intern wird der I2C – Bus benutzt, um das serielle EEPROM 24C65 mit dem Programm zu laden. Alle Routinen zur Bedienung des I2C-Busses sind hier natürlich bereits im ROM abgelegt. Doch das nützt nicht sehr viel, wenn man nicht weiß, wie die Einsprungadressen heißen. Wenn man jedoch über den Sourcecode verfügt, so kann man sich eine Vorstellung über die Abläufe verschaffen. Die I2C – Routinen sind mit dem folgenden Programm im ROM abgelegt:

```
;*********************************************************
;
;   C-Control Betriebssystem, 68HC05B6
;
;   Copyright (c) Conrad Electronic GmbH, Hirschau
;
;   Datei: i2c.asm
;
;          I2C-Bus Routinen
;
;          SCL an Port A0, SDA an Port A1
;          Achtung: nur Single-Master geeignet!
; 3.11.95
;*********************************************************

            psect bss,class=DATA

PORT  EQU   $00
DIR   EQU   $04
SDA   EQU   0
SCL   EQU   1

writebuf rmb 1

            psect text,class=CODE


;-----------------------------------------------
; void I2C_Init ( void );

     global     _I2C_Init
     signat     _I2C_Init,88
;-----------------------------------------------

; SCL und SDA ueber pull up high
; = bus not busy condition
; Mindestzeit warten
; ACHTUNG! nach power up sind SCL und SDA pulled up high
;          nach reset nicht unbedingt, also beide high schalten

_I2C_Init  BSET  SCL,DIR    ; SCL ist immer output (single master)
           BCLR  SCL,PORT   ; SCL lo, um SDA setzen zu koennen

           BSET  SDA,DIR    ; SDA output
           BSET  SDA,PORT

           BSET  SCL,PORT   ; clock
           BCLR  SCL,PORT   ;
           BSET  SCL,PORT   ; clock
```

```
           BCLR  SCL,PORT   ;
           BSET  SCL,PORT   ; clock
           BCLR  SCL,PORT   ;
           BSET  SCL,PORT   ; clock
           BCLR  SCL,PORT   ;
           BSET  SCL,PORT   ; clock
           BCLR  SCL,PORT   ;
           BSET  SCL,PORT   ; clock
           BCLR  SCL,PORT   ;
           BSET  SCL,PORT   ; clock
           BCLR  SCL,PORT   ;
           BSET  SCL,PORT   ; clock
           BCLR  SCL,PORT   ;

           BSET  SCL,PORT   ; scl hi

           RTS


;-------------------------------------------------
; void I2C_Start ( unsigned char devadr );

     global     _I2C_Start
     signat     _I2C_Start,4216
;-------------------------------------------------

; Startbedingung nach Init, Write oder Stop moeglich
; Vorbedingung: SCL lo (nach Write) oder
;               SCL hi und SDA hi (nach Init und Stop)
; SCL und SDA als Ausgang, SDA nach low ziehen
; = Startbedingung
; warten

_I2C_Start BSET  SDA,DIR    ; SDA hi ausgeben
           BSET  SDA,PORT   ; Vorbereiten der Startbed.

           BSET  SCL,PORT   ; SCL hi falls lo

           BCLR  SDA,PORT   ; SDA lo -> STARTBEDINGUNG
           BCLR  SCL,PORT   ;


           ; nach jeder Startbedingung wird sofort
           ; das Controlbyte geschrieben,
           ; also weiter bei I2C_Write

;-------------------------------------------------
; void I2C_Write ( unsigned char byte );

     global     _I2C_Write
     signat     _I2C_Write,4216
;-------------------------------------------------

_I2C_Write BSET  SDA,DIR         ; SDA out
           TXA              ;
           STA   writebuf   ; merken fuer retry
```

```
           LDX   #8         ; init loop
putnextbit ROLA
           BCC   lobit      ;
           BSET  SDA,PORT   ; hi bit
           BRA   writeclock
lobit      BCLR  SDA,PORT   ; lo bit

writeclock BSET  SCL,PORT   ; scl hi
           BCLR  SCL,PORT   ; scl lo

           DEX
           BNE   putnextbit ; loop

           ; ACK-Bit lesen
           ; Achtung: ACK ist low-aktiv

                BCLR  SDA,DIR         ; sda als Eingang
           BSET  SCL,PORT   ; scl hi
           BRSET SDA,PORT,retrywrite; no ACK -> retry
           BCLR  SCL,PORT
           RTS

           ; hierher nur bei control byte,
           ; wenn EEPROM noch nicht bereit

retrywrite BCLR  SCL,PORT   ;
           LDX   writebuf
           BRA   _I2C_Start


;-------------------------------------------------
; unsigned char I2C_Read ( void );

     global     _I2C_Read
     signat     _I2C_Read,73
;-------------------------------------------------

; Vorbedingung SCL lo
; SDA als Eingang
; 8 Bits vom Bus ueber Carry in Akku rollen

_I2C_Read  BCLR  SDA,DIR         ; sda als Eingang

           CLRA

           LDX   PORT
           RORX
           ROLA
           BSET  SCL,PORT   ; scl hi
           BCLR  SCL,PORT
           LDX   PORT
           RORX
           ROLA
           BSET  SCL,PORT   ; scl hi
           BCLR  SCL,PORT
           LDX   PORT
           RORX
```

```
            ROLA
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT
            LDX   PORT
            RORX
            ROLA
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT
            LDX   PORT
            RORX
            ROLA
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT
            LDX   PORT
            RORX
            ROLA
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT
            LDX   PORT
            RORX
            ROLA
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT
            LDX   PORT
            RORX
            ROLA
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT

                  ; nach jedem Byte-Read Acknowledge senden
                  ; Achtung: low-aktiv!

            BSET  SDA,DIR
            BCLR SDA,PORT    ; ACK

            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT   ; scl lo

            RTS


;-------------------------------------------------
; unsigned char I2C_ReadLast ( void );

      global     _I2C_ReadLast
      signat     _I2C_ReadLast,73
;-------------------------------------------------

_I2C_ReadLast   BCLR  SDA,DIR          ; SDA in

            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT   ; scl lo
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT   ; scl lo
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT   ; scl lo
            BSET  SCL,PORT   ; scl hi
```

```
            BCLR  SCL,PORT   ; scl lo
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT   ; scl lo
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT   ; scl lo
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT   ; scl lo
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT   ; scl lo

           ; ACK-FALSE senden

            BSET SDA,DIR     ; SDA out
            BSET  SDA,PORT   ; no ACK
            BSET  SCL,PORT   ; scl hi
            BCLR  SCL,PORT   ; scl lo


            ; STOP-Bedingung senden


;-----------------------------------------------
; void I2C_Stop ( void );

     global     _I2C_Stop
     signat     _I2C_Stop,88
;-----------------------------------------------

_I2C_Stop  BSET  SDA,DIR    ; SDA out
           BCLR  SDA,PORT   ; low

           BSET  SCL,PORT   ; scl high
           BSET  SDA,PORT   ; STOPBEDINGUNG
           RTS
```

Das Programm ähnelt der vorher entwickelten I2C – Routine. Die Leseroutine ist hier bei dem Systemprogramm sicherlich etwas umständlicher gelöst. Jetzt benötigt man noch die Einsprungadressen und es kann losgehen. In der Systemdatei Ccbasic.map findet man die nötigen Werte:

```
_I2C_Init          text    0811
_I2C_Read          text    086F
_I2C_ReadLast      text    08BB
_I2C_Start         text    083C
_I2C_Stop          text    08E5
_I2C_Write         text    0846
```

Es beginnt mit einem `I2C_Init`. Diese Arbeit hat das Programm schon für uns vorbereitet. Es ist ein „wildes“ Senden von Clockimpulsen, um den Chip aufzuwecken. Ich habe die Erfahrung gemacht, dass es auch ohne diese Impulse geht. Ein bisschen Alchemie ist eben überall zu finden. Auch das Datenblatt sagt nichts darüber aus.

`I2C_Read` ist der Lesebefehl mit anschließendem Acknowledge. Der gelesene Wert steht im Akku zur Verfügung.

I2C_Readlast besteht wieder aus „wilden“ Clockimpulsen und anschließendem No acknowledge, dem Befehl zur Signalisierung: kein weiteres Byte mehr lesen. I2C_Readlast endet mit dem Stoppbefehl.

I2C_Start leitet die Startbedingung ein, sendet das Adressbyte und holt das Acknowledge ab.

_I2C_Stop entspricht der Stoppbedingung.

Bleibt noch _I2C_Write. Diese Routine schreibt ein Byte über den I2C – Bus. An dem Eingangsbefehl txa (transfer x nach akku) ist erkennbar, dass der Schreibwert im X-Register übergeben werden muss. Anschließend wird automatisch das Acknowledge eingeholt. Jetzt sind wir fast so weit, dass das Programm entwickelt werden kann, das die Systemroutinen für den I2C – Bus benutzt. Bleibt vorab noch zu erklären, dass man vorsichtig mit der Benutzung des internen Busses sein muß. Der Chip lebt davon, dass er sein Programm ständig aus dem 24C65 EEPROM – Speicher holt. Dafür werden die internen Routinen genutzt. Nutzt man nun ohne Vorkehrungen diese Routinen, so kommt es unweigerlich zum Programmabsturz. Zunächst einmal muß der Speicherchip vom Bus getrennt werden. Nach der Abarbeitung der Routine muß er dann anschließend wieder richtig initialisiert werden. Dafür benötigt man noch die Adresse, in die der Speicherchip seine aktuelle Zeigerposition schreibt. Da 8 Kbytes verwaltet werden, wird dies in zwei Bytes, High und Low, vorgenommen. In der Dokumentation ist dies zu sehen.

```
_adrcounter   rbss    0066
```

Damit steht $066 (High) und $067 (Low) fest; die Zeiger zur Speicherzelle, die zuletzt im EEPROM angesprochen wurde. Diese Adresse wird zum Schluß in der Routine restore wieder hergestellt, so dass der interne Bus das Basicprogramm weiter abarbeiten kann.

```
*******************************************************
* Programmname: i2c_sys.asm
* Compilat    : i2c_sys.S19
* Basic       : i2c_sysd.BAS
* (c) Wolfgang Back
*******************************************************

adresse     equ $096   ; speichert die Chipadresse
i2c_wert    equ $094   ; speichert den Datenwert
richtung    equ $092   ; Lesen oder schreiben

ausgabe     equ $0A1    ; Ausgabe nach Basic

i2c_start   equ $083C  ; Start,schreiben,acknowledge
i2c_write   equ $0846  ; schreiben, acknowledge
i2c_read    equ $086F  ; lesen
i2c_last    equ $08BB  ; no acknowledge, stop
i2c_stop    equ $08E5  ; stop
adrhi       equ $066   ; Adresszeiger hi
adrlo       equ $067   ; Adresszeiger lo

*******************************************************

          org $101
          jsr i2c_last   ; Speicher vom Bus nehmen
          ldx adresse    ; Adresse    laden
```

```
          lda richtung   ; Lesen oder Schreiben
          bne lesen      ; <>0 dann lesen

schreiben jsr i2c_start  ; Adresse senden
          ldx i2c_wert   ; Schreibwert laden
          jsr i2c_write  ; Schreiben und getack
          jsr i2c_stop   ; vom Bus nehmen
          bra restore    ; verzweige immer

lesen     incx           ; Lesebit hinzufuegen
          jsr i2c_start  ; Leseadresse schreiben
          jsr i2c_read   ; Byte lesen
          sta ausgabe    ; ausgeben
          jsr i2c_last   ; Speicher vom Bus nehmen

restore   ldx #$A0       ; Systemadresse EEPROM (schreiben)
          jsr i2c_start  ; Adresse schreiben
          ldx adrhi      ; Adresscounter hi lesen
          jsr i2c_write  ; Hiadresse schreiben
          ldx adrlo      ; Adresscounter lo lesen
          jsr i2c_write  ; Lowadresse schreiben
          ldx #$0A1      ; Systemadresse  EEPROM (lesen)
          jsr i2c_start  ; Chip wieder anmelden
          rts
```

Damit steht eine effektive und kurze Routine zur Verfügung, die sogar zwei wertvolle Digitalports ersetzt. Der Bastler kann auf der Lallusplatine die beiden I2C – Ports 7 (clock) und 8 (data) durchtrennen und den I2C – Anschluß des Systems (pin 11 = SDA) und (Pin 12 = SCL) nutzen.

Ein kleines Demoprogramm zum Abtippen sei hier noch aufgeführt. Das Assemblerprogramm speichert sein Leseergebnis in die Zelle $0A1, was mit dem ersten Variablenbyte übereinstimmt. In Basic kann der Wert dann abgeholt werden. Die Definition `define ausgabe byte` muß daher immer an erster Stelle erfolgen. Es wird hier ein PCF8574 mit der Systemadresse `&H40` (alle Adresseingänge low) angesprochen. Zuerst wird ein Zufallswert gesendet, anschließend wird der Wert wieder gelesen. Die Geschwindigkeit dürfte überzeugen.

```
' Demoprogramm für I2C
define ausgabe byte
#main
randomize timer
sys &H0101 &H40,rand,0
sys &H0101 &H40,0,1
print ausgabe
goto main
syscode "I2C_SYS.S19"
```

Spezielle Änderungen des I2C – Busses

Mitunter ist es mit dem Schreiben und Lesen eines Bytes nicht getan. Zum Beispiel der DS1621 I2C – Temperaturchip, der im Kapitel „Messen von Temperaturen“ genauer beschrieben ist. Will man hier z.B. die Temperatur .5 Werten vornehmen, so müssen zwei Bytes nacheinander gelesen werden, ohne einen STOP – Befehl einzufügen. Das Assemblerprogramm kann daher einfach geändert werden. Das Beispiel zeigt, wie man

genauere Werte, z.B. 24.71 °C erhält. Zur Erklärung: zunächst wird das Register auf oneshot gesetzt. Dann wird eine Wandlung eingeleitet. Man wartet auf das Bit7=1 (Wandlung beendet). Dann liest man den Temperaturwert ohne Kommastelle. Danach wird der Counter $A8 und der Slopecounter $A9 ausgelesen. Beide Werte werden in die Formel Temperatur=gemessener Wert – 0.25 + (Counter – Slope) / Slope eingegeben. Da man hier keine Fließkommarechnung machen kann, wird die Berechnung der Nachkommastellen etwas komplizierter.

Zunächst das Basicprogramm:

```
'Progrannbane ds1621sy.bas

Define ausgabe byte
define counter byte
define temp    word
define wert    word
define slope   byte
define minus   bit[192]

#main
sys &H0101 &H90,&HAC,1,2     ' Oneshot ein
sys &H0101 &H90,&HEE,0,0     ' Wandlung ein
#nochmals
sys &H0101 &H90,&HAC,1,4     ' Controlregister lesen
if (ausgabe and 128) <> 128  then goto nochmals ' Bit 7 = 1 warten
sys &H0101 &H90,&HAA,0,0     ' Temperatur lesen einschalten
sys &H0101 &H90,0,0,3        ' Temperaturwert lesen
temp = ausgabe               ' Temperaturwert speichern
if (temp and 128)<>0 then minus=on else minus=off ' Bit 7 = 128?
sys &H0101 &H90,&HA8,0,4     ' Counter lesen
counter = ausgabe            ' Counter speichern
sys &H0101 &H90,&HA9,0,4     ' Slope lesen
slope = ausgabe              ' Slope lesen
if minus then print "- "; else print "+ ";
if minus then temp = -(256 - temp)  ' bei minus so rechnen
wert=(abs(temp*100-25)+((abs(slope*100-counter*100)/slope*100)/100)
* (1+(minus*2)))
print wert/100; ".";
if abs(wert mod 100) < 10 then print "0";
print abs(wert mod 100); " °C"
goto main
syscode "I2C_sysb.s19"
```

Das nachfolgende Assemblerprogramm hat einige Zusatzroutinen, mit denen man die Spezialitäten des Temperaturchips lesen und schreiben kann. Das Programm ist auskommentiert.

```
*******************************************************
* Programmname: i2c_sysb.asm
* Compilat    : i2c_sysb.S19
* Basic       : ds1621sy.BAS
* (c) Wolfgang Back
*******************************************************

adresse     equ $098   ; speichert die Chipadresse
i2c_wert1   equ $096   ; speichert den Datenwert
```

```
i2c_wert2   equ $094   ; speichert 2. Wert
richtung    equ $092   ; Lesen oder schreiben

ausgabe1    equ $0A1
ausgabe2    equ $0A2

i2c_start   equ $083C  ; Start,schreiben,acknowledge
i2c_write   equ $0846  ; schreiben, acknowledge
i2c_read    equ $086F  ; lesen
i2c_last    equ $08BB  ; no acknowledge, stop
i2c_stop    equ $08E5  ; stop
adrhi       equ $066   ; Adresszeiger hi
adrlo       equ $067   ; Adresszeiger lo

*******************************************************

          org $101

          jsr i2c_last   ; Speicher vom Bus nehmen

          ldx adresse    ; Adresse    laden

          lda richtung   ; Lesen oder Schreiben
          cmp #1
          beq lesen      ; verzweige nach lesen
          cmp #2
          beq schrei_2   ; zwei Bytes schreiben
          cmp #3
          beq lesen_2    ; zwei Bytes lesen
          cmp #4
          beq counter    ; Byte schreiben und lesen

schreiben jsr i2c_start  ; Adresse    senden
          ldx i2c_wert1  ; Schreibwert laden
          jsr i2c_write  ; Schreiben und getack
          jsr i2c_stop   ; vom Bus nehmen
          bra restore    ; verzweige immer

schrei_2  jsr i2c_start  ; Adresse    senden
          ldx i2c_wert1  ; Schreibwert laden
          jsr i2c_write  ; Schreiben und getack
          ldx i2c_wert2  ;
          jsr i2c_write  ;
          jsr i2c_stop   ; vom Bus nehmen
          bra restore    ; verzweige immer


lesen     incx           ; Lesebit hinzufuegen
          jsr i2c_start  ; Leseadresse schreiben
          jsr i2c_read   ; Byte lesen
          sta ausgabe1   ; ausgeben
          jsr i2c_last   ; Speicher vom Bus nehmen
          bra restore

lesen_2   incx           ; Lesebit hinzufuegen
          jsr i2c_start  ; Leseadresse schreiben
          jsr i2c_read   ; Byte lesen
```

```
          sta ausgabe1   ; ausgeben
          jsr i2c_read   ; 2. Byte lesen
          sta ausgabe2   ; ausgeben
          jsr i2c_last   ; Speicher vom Bus nehmen
          bra restore

counter   ldx adresse
          jsr i2c_start
          ldx i2c_wert1
          jsr i2c_write
          ldx adresse
          incx
          jsr i2c_start
          jsr i2c_read
          sta ausgabe1
          jsr i2c_last

restore   ldx #$A0       ; Programmspeicheradresse
          jsr i2c_start  ; Adresse schreiben
          ldx adrhi      ; Adresscounter hi lesen
          jsr i2c_write  ; Counter schreiben
          ldx adrlo      ; Adresscounter lo lesen
          jsr i2c_write  ; Counter schreiben
          ldx #$0A1      ; Leseadresse laden
          jsr i2c_start  ; Chip wieder anmelden
          rts
```

A / D - Wandler

Der Prozessor bietet uns weitere Spezialitäten. So können wir von Basic aus 8 analoge Eingänge benutzen, die z.B. mit Temperatursensoren oder anderen analogen Meßgebern versehen werden können.

Die acht Kanäle ad[1] bis ad[8] arbeiten mit einem einzigen AD-Wandler, der jeweils über eine Multiplexeinheit mit dem Analogwert des Kanals beschaltet wird. Danach wird die Wandlung eingeleitet und ein Statusbit abgefragt. Ist dieses Bit gesetzt, so kann der gewandelte Wert aus dem AD-Register gelesen werden. Auch dieses können wir sehr leicht mit dem Assembler bewerkstelligen. Dazu sollte wieder ein Blick in das Datenblatt des Prozessors Aufschluß geben.

Unter der Adresse $09 finden wir das A/D – Statusregister. Mit Bit 0 bis Bit 3 wird der Kanal gewählt. Bei der CControl kommen wir mit 3 Bits aus, da wir nur 8 Ports bedienen können.

Alle Kanäle (Bit 0 bis Bit 3) auf 0 bedeutet: wir schalten ad[1] auf den A/D - Wandler , entsprechend Bit 0=1, Bit 1 =1, Bit 2 =1 Bit3 =0 (dezimal 7) schaltet ad[8].

Bit 5 (ADON) = 1 schaltet die Wandlung ein.
Bit 7 (COCO) = 0 bedeutet Wandlung im Gange.
Bit 7 (COCO) = 1 bedeutet Wandlung beendet. (COCO = **Co**nversion **co**mplete)

4.5.3.1 A/D status/control register

	Address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	State on reset
A/D status/control	$0009	COCO	ADRC	ADON	0	CH3	CH2	CH1	CH0	0000 0000

ADON — A/D converter on

1 (set) – A/D converter is switched on; all port D pins act as analog inputs for the A/D converter.

0 (clear) – A/D converter is switched off; all port D pins act as input only pins.

Reset clears the ADON bit, thus configuring port D as an input only port.

Für die Daten der Wandlung existiert das zweite Register mit der Adresse $008. Ist die Wandlung beendet, so können hier die Daten abgeholt werden.

8.2.2 A/D result data register (ADDATA)

	Address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	State on reset
A/D data (ADDAT	$0008									0000 0000

ADDATA is a read-only register which is used to store the results of A/D conversions. Each result is loaded into the register from the SAR and the conversion complete flag, COCO, in the ADSTAT register is set.

Mit diesem Wissen können wir jetzt sehr leicht eine A/D – Wandlung mit dem Assembler vornehmen.

```
****************************************
* Ansteuerung des AD - Wandlers        *
* Programmname: ADWANDLE.ASM           *
* Compilat:     ADWANDLE.S19           *
* Basic:        define ausgabe byte    *
*               #main                  *
*               SYS &H101 kanal '(0-7)*
*               print ausgabe          *
*               goto main              *
*               SYSCODE "ADWANDLE.S19"*
****************************************
* A/D-Wandler - Register

addata  equ   $08    ; AD-Datenregister
adstat  equ   $09    ; AD-Stausregister

kanal   equ   $92    ; AD-Kanal aus Basic
ausgabe equ   $A1    ; Uebergabe nach Basic
```

```
        org   $101   ; Programmstart

        lda   #$20   ; Wandlung einschalten
        add   kanal  ; AD-Kanal addieren (0-7)
        sta   adstat ; in das Statusregister

Wait    brclr 7,adstat,Wait  ; warte auf Bit 7 (COCO)
        lda   addata         ; lade den AD-Wert
        coma                 ; invertiere Akku - Wert
        sta   ausgabe        ; Schreibe Wert nach Basic
        rts                  ; zurueck nach Basic
```

ADON ist das Bit 5 des Statusregisters. Es hat deshalb die Wertigkeit 32 oder $20. Je nach gewünschtem AD – Kanal wird das Register mit dem Wert `$20+kanal` gefüllt, die Wandlung beginnt. In der Warteschleife wird auf das COCO – Bit =1 gewartet. Bei beendeter Wandlung kann dann der Wert aus dem Datenregister in den Akku gelesen werden. Anschließend wird der gelesene Wert in das Userregister $A1 gespeichert, so dass Basic auf den Wert zugreifen kann.

Der Befehl **coma** ist hier nicht unbedingt vonnöten. Er zeigt lediglich, wie man Werte schnell invertieren kann. Beim Einsatz von Lichtsensoren (LDR) ist die Benutzung sinnvoll, da –mehr Licht- zu kleineren Widerstandswerten führt.

Wenn das Programm geladen und gestartet ist, so braucht man nur mit dem Finger den entsprechenden Analogport zu berühren, um im Terminalprogramm die sich ändernden Messwerte zu sehen.

D / A – Wandler

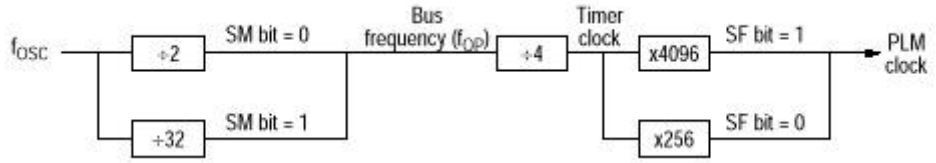
Oftmals benötigt man in der MSR – Technik Analogwerte, um z.B. die Drehzahl eines Motors zu steuern. Hat man nur digitale Signale zur Verfügung, so kann man dies durch eine Pulsweitenmodulation erreichen. Hierbei wird das digitale Signal mit einer bestimmten Frequenz getaktet. Das Analogsignal ergibt sich dann aus dem Signal – Pausenverhältnis. Sind die Pausen länger als die Signaldauer, so stellt sich ein geringer Analogwert ein, umgekehrt natürlich ein hoher Analogwert. Meistens glättet man den getakteten Ausgang mit einem Kondensator. Mitunter wünscht man aber auch gerade dieses digitale Taktsignal.

Der Prozessor hält hier zwei Register vor. PLMA ($0A) und PLMB ($0B), mit denen man eine Pulsweitenmodulation (PWM) erzielen kann. Auch hier gibt das Datenblatt Auskunft.

	Address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	State on reset
Pulse length modulation A (PLMA)	$000A									0000 0000
Pulse length modulation B (PLMB)	$000B									0000 0000

7.2 PLM clock selection

The slow/fast mode of the PLM D/A converters is selected by bits 1, 2, and 3 of the miscellaneous register at address $000C (SFA bit for PLMA and SFB bit for PLMB). The slow/fast mode has no effect on the D/A converters' 8-bit resolution (see Figure 7-3).



9.1.2 Miscellaneous register

	Address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	State on reset
Miscellaneous	$000C	POR[1]	INTP	INTN	INTE	SFA	SFB	SM	WDOG[2]	?001 000?

Zunächst einmal sehen wir, dass die DA-Register eine 8 – Bit - Länge besitzen, also 256 Werte auflösen können. Interessanter und etwas schwieriger wird es dann bei den nächsten Abbildungen.

Das Mehrzweckregister **misc** mit der Adresse $0C (12) kann mit Werten geladen werden, die sich auf die Ausgabegeschwindigkeit auswirken.
Bit 1 = 1 (SM) versetzt den gesamten Prozessor in den Slowmodus, d.h. alle Abläufe werden jetzt intern mit der Busgeschwindigkeit / 16 abearbeitet. Dieses wirkt sich dann auch auf die Baudrate der seriellen Schnittstelle aus. Die im Terminal gelieferten Werte sind daher unbrauchbar. Dieses Bit belassen wir daher auf 0.

Bit 2 (Wertigkeit 4) und Bit 3 (Wertigkeit 8) stellen den Takt der DA - Wandlung ein. Wenn beide Bits auf 1 gestellt werden, so wird sowohl PLMA und PLMB auf die langsame Taktgeschwindigkeit von 122 Hz eingestellt. Beide Bits auf 0 gesetzt taktet die Ausgänge mit 1923 Hz.

Für Testzwecke werden drei Einträge im **misc** – Register vorgenommen.

Der Eintrag $00 taktet schnell (ca 2 kHz),
$0C taktet langsam (122 Hz) und
$02 (Bit 1 = 1) schaltet in den Slowmodus (1/16 der Normaltaktfrequenz).

Wir können das sehr schön beobachten, wenn wir einen DA-Ausgang auf den Freuenzeingang DCF 77 legen. Zwei Krokodilklemmen genügen dazu. Da die Spannungswerte am Anfang zu gering sind, fängt die Variable bei 60 an.

Beim Zählerwert 130 wird in den langsamen Modus geschaltet.

Beim Zählerwert 200 wird der Slowmodus eingeschaltet. Er läuft bis 210 und schaltet dann wieder in den Normalmodus. Es erfolgt jetzt keine Anzeige mehr im Terminalprogramm, da die Oszillatorfrequenz um den Faktor 16 verringert wird und damit auch die serielle Schnittstelle nicht mehr die richtige Baudzahl liefert. Sie erkennen, dass der Slowmodus wirklich bedeutend langsamer ist.

Das Assemblerprogramm gestaltet sich dann recht einfach.

```
*******************************************
* D/A - Wandler Ansteuerung               *
* Programmname: DAWANDEL.ASM              *
* Compilat      DAWANDEL.S19              *
* Basic:        DAWANDEL.BAS              *
* (c) Wolfgang Back                       *
*******************************************

misc    equ  $0C    ; Mehrzweckregister
plma    equ  $0A    ; Pulslaengenmod. Ausgang A
plmb    equ  $0B    ; Ausgang B

eingabe equ  $94    ; Uebergaberegister
speed   equ  $92
        org  $101   ; Programmstart

        lda  speed  ; lade speed aus Basic
        sta  misc   ; misc - Register
        lda  eingabe; lade eingabe aus Basic
        sta  plma   ; schreibe nach A
        coma        ; Invertiere Akku
        sta  plmb   ; schreibe nach B
        rts         ; zurueck nach Basic
```

Die Timerregister

Für die Timerfunktionen sind eine Anzahl Register zur Verfügung. Sie sollen hier nicht alle beschrieben werden. Mit Hilfe des Datenblattes kann man hier vielleicht die eine oder andere Spezialität bearbeiten. Hier soll zunächst einmal gezeigt werden, wie man lange Pausen mit dem Timer realisieren kann.

Sie erinnern sich noch: Pausen im Assembler macht man normalerweise, indem man den Prozessor mit der Abarbeitung von Schleifen beschäftigt. Selbst wenn man zwei Schleifenzähler mit dem höchsten Wert 255 lädt, so erreicht man Pausenzeiten von weniger als einer Sekunde. Hat man nun eine Applikation, die mit Pausenzeiten von z.B. 10 Sekunden arbeiten soll, müsste man weitere Schleifen einbauen.

Hier wollen wir den Timer – Overflow betrachten und ihn für die Realisierung einer etwa zehn Sekunden langen Pause einsetzen.

Der Prozessor besitzt zwei Timer – Register, die jeweils 16 Bit breit sind, d.h. es sind zwei 8 Bit – Register mit dem bekannten Highbyte und dem Lowbyte. Der Timer wird ständig hochgezählt. Irgendwann ist natürlich das Highbyte mit 255 gefüllt und das 16 Bit-Register damit voll.

Beim nä chsten Takt wird ein Overflow erzeugt und beide Register werden wieder mit 0 gefüllt. Es geht dann immer so weiter. 16 Bit bedeuten 65535. 2 Mikrosekunden Taktgeschwindigkeit ergeben daher einen Overflow von ca. 130 Millisekunden.

Dieser Overflow wird in dem Timer – Status – Register (TSR $013) als TOF (**T**imer **O**ver**f**low) in dem Bit 5 gesetzt. Ein Lesen des Lowbytes des Counterregisters ($019) setzt das Bit 5 wieder zurück. Dieses können wir jetzt benutzen, um eine einfache lange Pause zu realisieren.

	Address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	State on reset
Timer status (TSR)	$0013	ICF1	OCF1	TOF	ICF2	OCF2				Undefined

	Address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	State on reset
Timer counter high	$0018									1111 1111
Timer counter low	$0019									1111 1100

```
**********************************
* Programmname: TIMER1.ASM       *
* Compilat:     TIMER1.S19       *
* Basic:        TIMER1.BAS       *
*               #main            *
*               sys &H0101       *
*               print second     *
*               goto main        *
*            syscode "timer1.s19" *
* (c) Wolfgang Back              *
**********************************

tsr         equ $13 ; Timerstatusregister
counterlow  equ $19 ; Counter Lowbyte
tof         equ 5   ; Bit 5 im Statusregister

            org $101; Programmstart

            lda #75 ; Schleifenzaehler auf 75

wait        brclr tof,tsr,wait ; warte auf overflow
            ldx counterlow     ; Lese Lowbyte = Reset von tof
            deca               ; abziehen

            bne wait           ; verzweige nach wait

            rts                ; zurueck nach Basic
```

Das, was oben bereits angesprochen wurde, ist in dem Programm realisiert. In der Warteschleife **wait** wird solange gewartet, bis **tof** vom Overflow gesetzt wurde. Danach wird mit **ldx counterlow** das Lowregister nur gelesen. Hiermit wird **tof** wieder auf 0 gesetzt.

Insgesamt wird dies 75 mal wiederholt. Wenn Sie nicht so lange warten wollen bis die Anzeige im Terminalprogramm wechselt, so können Sie natürlich jeden xbeliebigen Schleifenzaehler einsetzen (0 bis 255). Mit dem Basicprogramm und der Sekundenanzeige können Sie kontrollieren, wie genau der Assembler die Pause abarbeitet.