Externer Speicher

Serielles EEPROM 24C65

Die Zeiten haben sich wahrlich geändert. Wenn man an die frühen 80er Jahre denkt, so hatte man damals fast gar keinen Speicher zu Verfügung. Ein ZX81 von Sinclair war gerade einmal mit 1 Kilobyte RAM (Random Access Memory – löschbarer und neu beschreibbarer Speicher) ausgestattet. Geschicktes und ökonomisches Programmieren war angesagt. Eine sensationelle Leistung war es damals, das bekannte Spiel „Türme von Hanoi“ in den 1KB – Speicher zu programmieren.

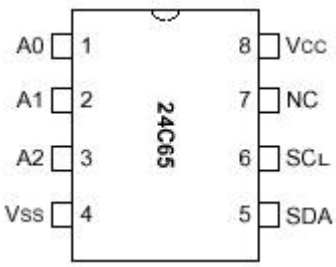
Nun ist die Arbeit mit der CControl auch nicht gerade eine Sache für Gigabytespezialisten. Doch immerhin sind hier 8 Kilobytes und noch ein paar Byte im Prozessor selbst zur Verfügung. Auf der CControlplatine befindet sich der externe Speicher in dem Chip 24C65.

Der 24C65 ist links als 8 - Pin - Chip erkennbar (weißer Pfeil).

Im Gegensatz zu den üblichen (parallel angesteuerten) EEPROM's ist der 24C65 ein seriell arbeitender Speicher. Der Name EEPROM ist abgeleitet von: Electrically Erasable Programmable Read Only Memory. Er besitzt 8 * 8 K (8*1024) Speicherzellen und ist mit dem schon bekannten I2C – Bus ausgestattet, über den die komplette Kommunikation läuft.

Normalerweise kann man den Chip dazu benutzen, um das Basicprogramm einzuladen und z.B. Messwerte oder andere Daten zu speichern. Den Speicherplatz im Chip muss man sich dann mit dem Basicprogramm teilen.

Mitunter macht es Sinn, die Daten außerhalb der CControl abzulegen, um sie anschließend wieder einzulesen. Vielleicht aber macht es auch nur Spaß, einmal mit einem externen Speicher zu experimentieren. Der problemlose Anschluß des kleinen Chips und sein Preis von ca. 6.- DM machen diese Experimente spannend.



Das Anschlussbild ist sicher leicht zu verstehen. A0, A1, A2 sind wieder die Adressierungspins für den I2C-Bus. Insgesamt 8 dieser Speicher könnten an einem I2C – Bus angeschlossen werden. Hier muss man natürlich einschränken. Der Chip auf der CControlplatine ist schon fest verdrahtet und daher fällt diese Adressierung aus.

Die Systemadresse des 24C65 ist fest eingestellt auf hex A0 oder dezimal 160.
Da alle A - Eingänge auf der Platine auf Masse liegen, ist diese Adresse vergeben. Mögliche Adressierungen also: 162, 164, 166, 168, 170, 172, 174.

Die bekannten seriellen Busleitungen SDA (Daten) und SCL (Clock) sowie VCC (+5 Volt) und Masse VSS sind im Anschlußbild zu erkennen. Das ist schon alles, was angeschlossen werden muß, so dass man es sich sogar leisten kann, den 8. Pin unbeschaltet zu lassen.

Im Laufe dieses Abschnitts werden wir verschiedene Möglichkeiten erarbeiten, wie man mit dem Chip kommunizieren kann.

Zunächst einmal muß man sich klarmachen, dass alle zu speichernden Daten einer strengen Ordnung unterworfen sind. Das A&O ist die Adressierung. 8192 Bytes (64 Kilobit) können abgespeichert werden. Das heisst, dass es daher auch 8192 Speicherplätze gibt.

Das, was an Platz 7 abgelegt wurde, muss später wieder von Platz 7 abgeholt werden. Damit wären wir bereits bei der wichtigsten Aufgabe:

1. Senden des Adressbytes (Chipadresse 160 + A - Eingangsadresse + Lesen/Schreiben)
2. Senden der gewünschten Adresse innerhalb des Speichers.

Ein Byte besteht bekanntlich aus 8 Bit, was 256 verschiedene Wertigkeiten bedeutet. Mit 256 Möglichkeiten können wir jedoch nicht alle 8192 Adressen erreichen. Ein zweites Byte wird hier eingesetzt, ein sogenanntes Highbyte und ein Lowbyte. Mit 2 Bytes können wir jetzt 65536 (256*256) Zustände darstellen, das reicht auf jeden Fall aus. Bei einer 8 – Bit orientierten Elektronik (CControl und 24C65) hat man oft mit diesen Umrechnungen zu tun, wenn man grössere Werte darstellen will. Um die Handhabung der beiden Bytes zu verstehen, habe ich die Zerlegung hier einmal in Basic realisiert.

Der Wert des Highbytes ist einfach mit 256 zu multiplizieren, danach wird der Wert des Lowbytes addiert. Umgekehrt geht es genauso: Der Wert wird durch 256 dividiert, das ganzzahlige Ergebnis ergibt das Highbyte. Der Rest ergibt das Lowbyte.

```
Wert = 1000
Highbyte = Wert / 256
Lowbyte = Wert MOD 256
```

Ergebnis:

Highbyte = 3
Lowbyte = 232.

Umgekehrt geht es dann genauso:

Wert = Highbyte * 256 + Lowbyte

3 * 256 = 768
768 + 232 = 1000

Bekanntlich führen viele Wege nach Rom. Man könnte es auch etwas komplizierter programmieren. Hier nur als Beispiel gedacht, wie man 16 Bit auch in zwei Werte auftrennen könnte.

```
for i = 15 to 0 step -1
if i>7 and adresswert and (1 shl i) then adrhi=adrhi + (1 shl (i-8))
```

```
if i<8 and adresswert and (1 shl i) then adrlo=adrlo + (1 shl i)
next i
```

Die oberen 8 Bit der 16 Bit langen Variablen **adresswert** werden mit ihrer Wertigkeit verglichen und das Highbyte **adrhi** errechnet, danach sind die unteren 8 Bits dran. Das Ergebnis steht dann in **adrhi** und **adrlo**.

Damit können wir jetzt die Anforderung präziser umsetzen:

1.) Senden des Adressbytes (Chipadresse 160 + A - Eingangsadresse + Lesen/Schreiben)
2.) Ermitteln des Highbytes und senden
3.) Ermitteln des Lowbytes und senden.

In meinem Versuchsaufbau war der externe Speicher mit der Adresse A0=high, A1=low und A2=low versehen. Dieses ergibt die Chipadresse 160 + 2 = 162 oder hex A2.
Die ersten Programmschritte sehen daher so aus:

```
Define adresswert word
Define adrhi      byte
Define adrlo      byte
Define i2c_ein    byte
Define i2c_aus    byte
Define adresse &HA2

#main
print „Adresse eingeben“
input adresswert
adrhi = adresswert / 256
adrlo = adresswert mod 256

gosub  start

i2c_ein = adresse
gosub  putbyte
gosub  getack

i2c_ein = adrhi
gosub  putbyte
gosub  getack

i2c_ein = adrlo
gosub  putbyte
gosub  getack
```

Dieser Vorspann ist für alle Operationen gleich, sowohl für Lese- wie für Schreiboperationen. Im folgenden unterscheiden sich die beiden Routinen. Zunächst soll ein Byte gelesen werden. Mit einer neuen Startbedingung wird ein `gosub stop` überflüssig.

```
gosub start
I2c_ein = adresse + 1   ' Lesen
gosub putbyte
gosub getack
gosub getbyte
gosub givenoack
print "Speicherzelle " adresswert; " = " ; i2caus
```

Das Unterprogramm `givenoack` enthält zum einen die Signalisierung, daß kein weiteres Byte mehr gelesen werden soll, zum anderen verzweigt es in die Stoppbedingung. Die Aktion ist beendet.

Beim Schreibvorgang ist dieses ähnlich:

```
gosub start
i2cein = adresse ' Schreiben
gosub putbyte
gosub getack

print "Schreibwert eingeben: "
input i2c_wert
gosub putbyte
gosub getack
gosub stop
```

Damit ist es jetzt möglich, eine Adresse innerhalb des 24C65 auszuwählen und einen 8 – Bitwert einzutragen. Der Wert bleibt nun mindestens 10 Jahre gespeichert, auch wenn man die Versorgungsspannung wegnimmt. Mitunter möchte man mehrere Werte gleichzeitig lesen / schreiben – oder besser: nacheinander lesen / schreiben. Auch dieses ist leicht möglich.
Der Vorspann bleibt gleich. Die Leseroutine für 10 Bytes sieht dann so aus:

```
gosub start
I2c_ein = adresse + 1  ' Lesen
gosub putbyte
gosub getack

for i=1 to 10
gosub getbyte
if i < 10 then gosub giveack else gosub givenoack
print "Speicherzelle " adresswert+i; " = " ; i2caus
next i
```

Wir sehen an der Zeile mit der `if < 10` – Bedingung, dass nur beim letzten Byte die Unterroutine `givenoack` aufgerufen wird, die dem Chip mitteilt: letztes Byte gelesen, `stop` – Bedingung einleiten.

Auch beim Schreibvorgang können wir mehrere Bytes nacheinander schicken, ohne jeweils die Anfangsroutine zu senden. Ein interner Zeiger im Chip erhöht jeweils die Adresse, so daß das gesendete Byte an der richtigen Stelle gespeichert wird.

```
gosub start
i2cein = adresse ' Schreiben
gosub putbyte
gosub getack

for i= 1 to 50
i2c_ein = i
gosub putbyte
gosub getack
next i
gosub stop
```

Es gibt noch weitere Möglichkeiten, um die Bytes in den Speicher zu bringen und sie später auszulesen. Es würde jedoch den Rahmen sprengen, wenn man diese Möglichkeiten alle genau erklären würde. Im Datenblatt auf der CD kann man diese Möglichkeiten nachlesen und selbst damit experimentieren. Hier soll es genügen, dass wir mit den aufgezeigten Routinen den Speicher vollständig bedienen können.

Wenn man sich die Dateioperationen in Basic ansieht, so kann man jetzt sicherlich besser verstehen, wie sie funktionieren. Hier gibt es die Dateibefehle.

`open# for write`, `open# for read`, `open# for append`, `print#` und `input#` und `close#`.

`Open# for write` bewirkt in Basic, dass alte Dateiinhalte überschrieben werden und die Speicherung der eingegebenen Werte am Anfang der Datei beginnt. In unserem Falle würden wir hier einfach auf die Adresse 0 verweisen und dort mit der Speicherung beginnen. In der Praxis jedoch wird man wahrscheinlich erst bei Adresse 3 anfangen. Wir sehen gleich, warum das so ist.

Ein `open# for append` wirkt etwas anders. Der Speicherinhalt bleibt erhalten und das neu zu speichernde Byte wird hinter das zuletzt gespeicherte Byte gesetzt. Die Adresse, in der das zu speichernde Byte abgelegt werden soll, merkt man sich normalerweise in den ersten zwei Bytes am Speicheranfang. Adresse 0 – Highbyte, Adresse 1 – Lowbyte. Bei einem neuen Appendbefehl werden daher die ersten beiden Bytes gelesen, der Wert des Adresszeigers um 1 erhöht und dann an der richtigen Stelle abgespeichert.

Mit `close#` wird eine Dateioperation abgeschlossen. Das ist der Moment, in dem man den Adresszeiger in die ersten beiden Bytes abspeichert. Will man in der Anwendung mit den Append – Möglichkeiten arbeiten, so muss man die gesendeten Bytes also immer mitzählen, damit der Adresszeiger immer mit seiner aktuellen Position abgespeichert werden kann.
Hat man Anwendungen, die immer mit einem ‚frischen' Dateiinhalt beginnen, so kann man darauf verzichten.

Mit `open# for read` wird eine Leseoperation eingeleitet, d.h. es wird bei der Übermittlung der Systemadresse das Lesebit gesetzt.

`Print#` entspricht daher einem `putbyte`, `input#` entspricht dem `getbyte` und der `close#` - Befehl findet sich im I2C-Programm als `stop` oder `getnoack` wieder.

Man wird bei der Speicherung über den I2C-Bus feststellen, dass der Lese- wie der Schreibvorgang relativ langsam abläuft. Zum einen ist es der zeitaufwendige I2C-Bus selbst, zum anderen kommt noch der 24C65 hinzu, der für jedes zu speichernde Byte einen Brennvorgang auslösen muss, der um die 10msec dauert.

Um zumindest die Kommunikation mit dem Speicher zu beschleunigen, kann hier wieder ein Assemblerprogramm helfen.

Vorab sei noch erklärt, dass man mit dem SYS – Befehl aus Basic dem Assemblerprogramm Werte übergeben kann. Diese werden dann in der CControl in bestimmten Registern abgeholt. Ein `SYS &H0101 adresse` z.B. führt dazu, dass der 8 Bit-Wert im Register `$092` abgelegt wird. Bequem ist die Abspeicherung größerer Werte als `255`. Wird z.B.
`SYS &H0101 1000` übertragen, so wird der Wert `1000` in ein Hibyte und ein Lobyte zerlegt. Das Hibyte mit dem Wert `3` finden wir dann an der Adresse `$091`, das Lobyte mit dem Wert `232` an der Adresse `$092`.

Man muss noch wissen, dass bei der Übergabe mehrerer Parameter die erste Speicherstelle jeweils um 2 nach oben geschoben wird. In dem untenstehenden Programm werden 4 Werte mit dem `Sys` - Befehl übergeben. Daher teilt sich der Speicherbereich so auf:

```
Sys adresse, adresszeiger, wert, richtung
```

`richtung` ist ein Byte, es wird zuletzt gesendet und erhält daher den ersten Speicherbereich für das Lowbyte. `$092` beinhaltet den Wert von `richtung`, `$091` bleibt frei und könnte vom Assemblerprogramm anderweitig benutzt werden.

`adresszeiger` ist eine Wordvariable und belegt `$093` mit dem Hibyte und `$094` mit dem Lobyte.

`wert` und `richtung` sind wieder Einbytevariablen und sind an den Adressen $096 und `$098` abholbar. `$095` und `$097` werden im Assemblerprogramm als Hilsregister genutzt.

Diese Methode der Parameterübergabe geht nur mit einer neueren Software. Das neuere Programm CCEW32D.EXE ist auf der CD vorhanden. Man kann die Parameterübergabe auch noch anders realisieren, wobei man dann jedoch das Hibyte und das Lobyte selbst errechnen muss.

Man deklariert dann die Variablen in Basic und holt sie an den 24 reservierten Variablenadressen ab. Der Userbereich beginnt hier ab Speicherzelle `$0A1`. Eine Deklaration würde dann so aussehen. Entsprechend der Daklarierung werden die Speicherzellen vergeben.

```
define adresse  byte ' entspricht $0A1
define adrhi    byte ' entspricht $0A2
define adrlo    byte ' entspricht $0A3
define wert     byte ' entspricht $0A4
define richtung byte ' entspricht $0A5
```

Das Assemblerprogramm ist ähnlich wie das Basicprogramm aufgebaut.

```
*****************************************************
* Assemblerprogramm zur Kommunikation mit einem
* SEEPROM 24C65
* Programmname 24C65.ASM
* Compilat     24C65.S19
* BASIC        24C65.BAS
*****************************************************

bport       equ $01              ; Digitalports 1 - 8
bpdir       equ $05              ; Richtung port B
data        equ 7                ; Dataport (Port 8)
clock       equ 6                ; Clockport(Port 7)
adresse     equ $098             ; aus Basic
adrhi       equ $095             ; Adresszeiger high
adrlo       equ $096             ; Adresszeiger low
i2c_wert    equ $094             ; Schreibwert
richtung    equ $092             ; Lesen oder schreiben
ausgabe     equ $0A1           ; Basic Useradresse
merker      equ $095           ; RAM - Variable
wert        equ $097           ; RAM - Variable
*************************************************
```

```
          org   $101            ; Startadresse

          bset  clock,bpdir     ; clock als -Ausgang-

main      jsr   start           ; Startbedingung
          lda   adresse         ; Adresswert laden
          jsr   putbyte         ; Adresse schreiben
          jsr   getack          ; warten auf acknowledge
          lda   adrhi           ; Highwert abholen
          jsr   putbyte         ; schreiben
          jsr   getack          ; warten auf acknowledge
          lda   adrlo           ; Lowert abholen
          jsr   putbyte         ; schreiben
          jsr   getack          ; warten auf acknowledge

          lda   richtung        ; lade Richtungsbyte
          cmp   #1              ; vergleiche Wert
          beq   lesen           ; verzweige nach lesen wenn 1

schreiben lda   i2c_wert        ; lade Adresse
          jsr   putbyte         ; schreibe Adresse
          jsr   getack          ; warten auf acknowledge
          jsr   stop            ; Stopbedingung -Ende-
          rts                   ; nach Basic

lesen     jsr   start           ; neue Startbedingung
          lda   adresse         ; lade Adresse
          inca                  ; setze Lesebit
          jsr   putbyte         ; Leseadresse schreiben
          jsr   getack
          jsr   getbyte         ; hole den Lesewert
          jsr   givenoack       ; beende Lesevorgang
          jsr   stop            ; stop -Ende-
          rts                   ; nach Basic

start     bset  data,bpdir      ; data als -Ausgang-
          bset  data,bport      ; data  1 setzen
          bset  clock,bport     ; clock 1 setzen
          bclr  data,bport      ; data  0 setzen
          bclr  clock,bport     ; clock 0 setzen
          rts

stop      bset  data,bpdir      ; data als -Ausgang-
          bclr  data,bport      ; data  0 setzen
          bset  clock,bport     ; clock 1 setzen
          bset  data,bport      ; data  1 setzen
          rts                   ; zurueck nach Basic


putbyte   ldx   #8              ; Shiftzaehler laden
          bset  data,bpdir      ; data als -Ausgang-
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

getbyte   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 Siftvariable
          lsra                  ; logisch rechts shiften
          sta   wert            ; speichern
          jsr   pulse           ; clock senden
          decx                  ; Schleife um 1 verkleinern
          bne   loop            ; verzeige nach loop, wenn <>0
          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
          bclr  data,bpdir       ; data als Eingang
warte     brset data,bport,warte ; auf data 0 warten
          jsr   pulse
          rts


giveack   bset  data,bpdir       ; data als -Ausgang-
          bclr  data,bport       ; data auf low
          jsr   pulse            ; pulsen
          bset  data,bport       ; data auf high
          rts

givenoack bset  data,bpdir       ; data als -Ausgang-
          bset  data,bport       ; data auf high
          jsr   pulse            ; pulsen
          rts
```

Mit dieser Routine sollte der Datenverkehr zwischen CControl und 24C65 bedeutend schneller ablaufen. Natürlich gehört noch ein Basicprogramm dazu. Hier ein ein ganz einfach gestaltetes Programm, das den Speicher ständig ausliest, beim Overflow dann von vorne wieder anfängt.

```
define ausgabe byte ' Liegt auf $0A1
define wert    byte ' Schreibwert
define i       word ' Wordvariable für Adresszeiger
wert = 65
#main
sys &H0101 &HA2, i, 65, 1
```

```
print ausgabe; " ";
i = i + 1
goto main

syscode "24C65.s19"
```

Wenn das Programm in dieser Weise läuft, so wird der Speicher von der Adresse 0 an gelesen. Ändert man den letzten Parameter 1, der für Lesen steht in eine 0, so wird der Chip mit `wert,` vollgeschrieben, in obigem Falle mit 65.

Beim Laden des Programmes wird man erkennen, dass es sehr lange dauert, bis die Assemblerdatei eingelesen ist. Wenn man anschließend mit dem Basicprogramm experimentieren möchte, so ist es sinnvoll die Assemblerroutine im internen Speicher zu belassen. Sie braucht nicht jedesmal neu geladen werden. Wenn man sie mit einem Hochzeichen ‚ausremt', so wird nur das kleine Basicprogramm geladen. Erst wenn man im Assembler wieder etwas ändert, muß natürlich die Datei neu eingelesen werden.

Wenn man die Systemressourcen des Programmes kennt (ich bekam glücklicherweise das Programmlisting in die Hand), so kann man noch effektiver programmieren. Ich möchte an dieser Stelle eine Hinführung für Spezialisten und solche, die es werden wollen, dokumentieren. Da sich ein 24C65 als Programmspeicher auf der Platine befindet, muss natürlich auch der I2C-Bus im System realisiert sein. Wenn man weiß, wo die Routinen gespeichert sind, so kann man effektiver Assemblerprogramme schreiben, indem man die eingebauten Routinen nutzt.

Ich muß allerdings dazu sagen, dass ich mir beim Experimentieren oftmals das Programm zerschossen habe, weil man schnell in Konflikte mit dem 24C65 auf der Platine kommen kann, das ja das Basicprogramm beherbergt.

Jedenfalls kann man nichts kaputtmachen – ein erneutes Laden des Programms bringt alles wieder ins Lot.

Bei der Konstruktion von Lallus haben wir zwei Digitalports für einen eigenen I2C-Bus geopfert, obgleich SDA und SCL auf Pins herausgeführt sind. Diese Pins kann man natürlich nutzen, wenn man das entsprechende Spezialwissen hat. Sie können sich vielleicht erinnern, dass Sie in der Bedienungsanleitung gelesen haben , SDA und SCL * für spätere Anwendungen reserviert.'

Wenn Sie es sich zutrauen, die beiden Pins SDA (11 auf Steckleiste 2) und SCL (12 auf Steckleiste 2) anzuzapfen und sie mit dem I2C-Bus von Lallus zu verbinden, so können Sie die weiter unten erklärten Routinen benutzen.

Sie gewinnen dann zwei zusätzliche Digitalports, wenn Sie alle I2C-Funktionen mit den Systemroutinen erledigen. Wenn Sie die Lallusplatine benutzen, so sollten Sie die Verbindungen zum I2C-Bus (Port 8 und Port 7) auftrennen. Wohlgemerkt: nur der sollte das tun, der später evtl. auch alles rückgängig machen kann.

Fangen wir an.

Alle Systemressourcen sind irgendwo in der CControl gespeichert. Im festen ROM-Bereich gibt es daher auch Adressen, die man über den Assembler anspringen und daher nutzen kann. Der I2C-Bus ist mit wenigen multifunktionalen Routinen realisiert.
Ein `I2C_start` schreibt die Startbedingung und das Adressbyte weg, holt das acknowledge. Ein `i2C_Last` meldet ‚nicht mehr lesen' und stop. Im Programm sind die Einsprungadressen für die verschiedenen I2C-Funktionen ersichtlich. Die Auskommentierung gibt Aufschluss über die Funktionen.

Wichtig ist, daß man den Chip 24C65 mit dem Basicprogramm ordnungsgemäß vom Bus abmeldet und ihn nach der Aktion wieder anmeldet, damit das Basicprogramm weiter arbeiten kann. Wie oben bereits beschrieben, speichert man den Adresszeiger mit Hibyte und LoByte in einem Register ab. Hier ist es $066 und $067. Diese Werte muss man dem Chip wieder liefern, wenn man sich vom Bus abgemeldet hat.

```
*******************************************************
* Programmname: 24c65sys.asm
* Compilat      24C65sys.s19
* Basic         24c65.bas
*******************************************************
adresse     equ $098   ; speichert die Chipadresse
i2c_werthi  equ $095   ; speichert die Adresse hi
i2c_wertlo  equ $096   ; speichert die Adresse low
i2c_wert    equ $094   ; speichert den Schreibwert
richtung    equ $092   ; Lesen oder schreiben

ausgabe     equ $0A1

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   ; wo steht der Basiczeiger hi
adrlo       equ $067   ; wo steht der Basiczeiger lo
*******************************************************

          org $101

          jsr i2c_last   ; Speicher vom Bus nehmen

          ldx adresse    ; Adresse    laden
          jsr i2c_start  ; Adresse    senden
          ldx i2c_werthi ; Adressbyte hi lesen
          jsr i2c_write  ; Adressbyte hi schreiben
          ldx i2c_wertlo ; Adressbyte lo lesen
          jsr i2c_write  ; Adressbyte lo schreiben

          ldx adresse    ; liest die Chipadresse

          lda richtung   ; Lesen oder Schreiben
          cmp #1         ; pruefe auf 1
          beq lesen      ; verzweige nach lesen

schreiben ldx i2c_wert   ; lade i2c_wert
          jsr i2c_write  ; schreibe
          jsr i2c_last   ; nicht mehr lesen, stop
           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       ; Speicheradresse laden
            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
```