1.

Assembler-Tricks für den 68HC11

von Oliver Thamm

1.1. Über die Kunst, ein Byte herbeizuzaubern

Die geschickte Ausnutzung vorhandener Ressourcen, das Sparen von Speicherplatz und die Minimierung von Ausführungszeiten war das Bestreben der Programmierer seit der Erfindung des Bits. Das Ringen um Mikrosekunden, der Kampf um das letzte Byte Programmspeicher und die kleinste Ecke RAM hat aber nicht nur akademischen Reiz, auch in der Praxis ergibt sich oft die Notwendigkeit, aus weniger mehr zu machen. Oft ist es so, daß nur 95 Prozent der Forderungen an ein Programm in 100 Prozent des vorgesehenen Speichers passen - zumindest im ersten Anlauf. Das ist ärgerlich, fordert aber auch heraus, einmal genauer hinzuschauen. Was in einer solchen Situation akuten Speichermangels zu tun ist, soll Gegenstand des folgenden Beitrags sein.

1.2. Die Zero Page

Der HC11 hat lediglich zwei Akku-Register und zwei Index-Register, mit denen der Anwender etwas unternehmen kann. Andere Mikrocontroller oder -prozessoren spendieren an dieser Stelle mehr Ressourcen. Dieser "Mangel" wird mehr als ausgeglichen durch die Möglichkeit, auf die ersten 256 Byte des Adreßraumes mit einer speziellen, sehr schnellen Adressierungsart zuzugreifen.

Diese Adressierungsart wird beim HC11 als Direct Addressing bezeichnet. Die Schnelligkeit dieser Zugriffsmethode ergibt sich durch die Einsparung des höherwertigen Bytes der Adresse, welches ja im genannten Adreßbereich stets den konstanten Wert $00 aufweist (daher stammt auch der Begriff "Zero Page"). Bei einem gängigen Befehl wie

```
LDAA LOCATION
```

kann man immerhin ein Drittel des Maschinencodes einsparen, indem man statt der (normalen) Extended Adressierung die verkürzte Direct Adressierung verwendet. Ein Beispiel für die beiden Varianten (Assemblerlisting):

```
1: 0080            ZEROPB  EQU $0080
2: 0000
3: 0000                    ORG $B600
4: B600
5: B600 B60080             LDAA ZEROPB
6: B603 B70080             STAA ZEROPB
7: B606 9680               LDAA <ZEROPB
8: B608 9780               STAA <ZEROPB
```

Die Erkenntnis, auf diese Weise u.U. pro Befehl ein Byte zu sparen, ist an sich trivial. Dennoch lohnt es sich, ein Assemblerprogramm bei aufkommendem Platzmangel auf diesen Aspekt hin nochmals aufmersam durchzusehen. Manchmal gelingt es auch, durch Umordnen von RAM-Speicherbereichen die eine oder andere Direct Adressing Möglichkeit auszunutzen. Man muß z.B. nicht unbedingt einen zusammenhängenden Pufferbereich bei $0000 beginnen lassen, wenn sich noch weitere Variable im Adreßraum $0100 und höher anschließen. Bei einer solchen Konstellation empfiehlt es sich, Puffer und Variablen gegeneinander zu verschieben. Der Grund dafür liegt in der Möglichkeit, die Variablen, welche dann im Zero Page Bereich liegen, mittels Direct Addressing speicherplatzsparend anzusprechen, wohingegen Pufferbereiche bevorzugt über ein Indexregister adressiert werden, wobei es gleich ist, ob der Puffer dann in der Zero Page oder an einer anderen Stelle im Speicher liegt.

1.3. Indizierte Adressierung

Der HC11 besitzt zwei Indexregister X und Y. Mit diesen 16-Bit breiten Indexregistern kann man den gesamten 64 KB Adreßraum des HC11 überstreichen. Um ein Byte mittels indizierter Adressierung (Indexed Addressing) anzusprechen, muß man zuerst eines der Indexregister mit der gewünschten Adresse laden. Dazu "opfert" man zunächst drei Byte / drei Taktzyklen (Index Register X) bzw. vier Byte / vier Taktzyklen (Index Register Y). Anschließend jedoch kann man pro Zugriff ein Byte einsparen, wenn man Indexed Adressing mit einem normalen Zugriff via Extended Addressing vergleicht. Die Rechnung geht also auf, wenn mehr als drei Instruktionen mittels X-Indizierter Adressierung verkürzt werden können. Die Verwendung des Y-Registers erscheint in diesem Zusammenhang aussichtslos, da man jeden Einsatz mit einem zusätzlichen Prebyte (z.B. wird aus dem Zwei-Byte-Code A600

für LDAA 0,X ein Drei-Byte-Code 18A600 für LDAA 0,Y) erkauft. Dieses Prebyte macht die Einsparung wieder zunichte - im Rahmen dieses Beitrags betrachten wir also das Indexregister Y nicht weiter.

Das folgende Beispiel soll der Illustartion dienen. Variante A ist stellt den klassischen Weg dar:

```
 1: 2000            BUFFER  EQU $2000
 2: 0000
 3: 0000 B62000             LDAA BUFFER
 4: 0003 F62001             LDAB BUFFER+1
 5: 0006 B72001             STAA BUFFER+1
 6: 0009 F72000             STAB BUFFER
```

Variante B kommt Dank indizierter Addressierung mit einem Byte weniger aus:

```
 1: 2000            BUFFER  EQU $2000
 2: 0000
 3: 0000 CE2000             LDX #BUFFER
 4: 0003 A600               LDAA 0,X
 5: 0005 E601               LDAB 1,X
 6: 0007 A701               STAA 1,X
 7: 0009 E700               STAB 0,X
```

Besondere Bedeutung erlangt die indizierte Adressierung beim Zugriff auf die Steuerregister des HC11. Diese Steuerregister befinden sich alle im Bereich $1000 bis $103F (bis $105F beim HC11F1). Eine gängige Arbeitsweise ist es, das X-Register mit der Basisadresse dieses Register Areas zu laden und dadurch einen schnellen Zugriff auf alle Register zu ermöglichen. Das folgende Beispiel initialisiert das Asynchrone Serielle Interface (SCI) unter Zuhilfenahme des indizierten Zugriffs auf die Steuerregister:

```
 1: 1000            REGBAS  EQU     $1000
 2: 002B            xBAUD   EQU     $2B
 3: 002D            xSCCR2  EQU     $2D
 4: 0000
 5: 0000 CE1000     initSCI LDX     #REGBAS
 6: 0003 8630               LDAA    #$30
 7: 0005 A72B               STAA    xBAUD,x
 8: 0007 860C               LDAA    #$0C
 9: 0009 A72D               STAA    xSCCR2,x
10: 000B 39                 RTS
```

Die obige Routine verwendet zwar konsequent Indexed Addressing, ist aber dennoch noch nicht überzeugend, da bei dieser konkreten Implementierung keine Einsparung von Speicherplatz erfolgt. Erst bei einer Weiterentwicklung der Idee wird die Vorgehensweise lukrativ: Was wäre, wenn man in der Initialisierungsphase des Programms einmalig das Indexregister X mit dem Wert REGBAS lädt, und dann an jeder Stelle des Programms darauf zurückgreift? Schlagartig verlieren alle Zugriffe auf Steuerregister an Masse. Da derlei Zugriffe häufig vorkommen, wird man auch eine nicht unbeachtliche Anzahl von Programmbytes einsparen können. Natürlich hat auch diese Vorgehensweise einen Haken. Der besteht darin, daß das X-Register für anderweitige Verwendung blokkiert ist, es sei denn, man rettet es vor einer Änderung auf den Stack und führt nach der Benutzung eine Reinitialisierung durch, indem man es vom Stack zurtückholt. Aber Vorsicht! Allzuleicht vertraut man auf die - fast - konstante Belegung des X-Registers. Rechnet man aber innerhalb einer Interrupt-Service-Routine nicht mit der temporären Umbelegung des Indexregisters, kann man unschöne Überraschungen erleben. Das folgende Programmbeispiel zeigt eine solche Konfliktsituation:

```
 1: 1000            REGBAS  EQU     $1000
 2: 0000            xPORTA  EQU     $00
 3: 0004            xPORTB  EQU     $04
 4: 0000
 5: 0000                    ORG $B600
 6: B600
 7: B600 CE1000     main    LDX     #REGBAS
 8: B603 8600               LDAA    #0
 9: B605 4C         loop    INCA
10: B606 BDB60B             JSR     sub1
11: B609                    ; ...
12: B609 20FA               BRA     loop
13: B60B
14: B60B 3C         sub1    PSHX
15: B60C CE2000             LDX     #$2000  ; Crit
16: B60F E600               LDAB    0,X     ; Crit
17: B611                    ; ...
18: B611 38                 PULX            ; Crit
19: B612 39                 RTS
20: B613
21: B613 A600       isr     ldaa    xPORTA,x
22: B615 8401               anda    #$01
23: B617 A704               staa    xPORTB,x
24: B619 39                 rts
```

Zu Beginn wird hier X ordnungsgemäß auf $1000 gesetzt. Wird irgendwann durch ein externes Ereignis ein Interrupt ausgelöst, der die Interrupt-Service-Routine isr aufruft, geht alles glatt, solange sich das Hauptprogramm zur Zeit des Interrupts nicht gerade im kritischen Bereich Crit aufhielt. Das kann aber nicht ausgeschlossen werden, wodurch es sporadisch zu Fehlfunktionen kommen wird. Wenn Ihre Applikation nicht immer, aber immer öfter abstürzt, und Interrupts eine Rolle spielen - vielleicht erinnern Sie sich...

1.4. Double Accu D

In Initialisierungssequenzen kommt es oft vor, daß ein Akkuregister nacheinander mit verschiedenen Werten geladen wird und der Inhalt dann zu Steuerregistern oder Speicherbereichen transferiert wird. Das folgende Beispiel zeigt ein solches Codefragment:

```
1: 2000            ABC     EQU     $2000
2: 2010            XYZ     EQU     $2010
3: 0000
4: 0000 8622               LDAA #$22
5: 0002 B72010             STAA XYZ
6: 0005 865B               LDAA #$5B
7: 0007 B72000             STAA ABC
```

Ein Byte läßt sich in obigem Beispiel durch Verwendung des Double Accu D einsparen. D belegt bekanntlich den selben internen Speicherplatz wie die beiden 8-Bit Akkus A und B. Das höherwertige Byte von D ist identisch mit A, das niederwertige Byte entspricht B. Manchmal verliert man diesen Umstand aus den Augen, das kann gefährlich sein. Bei Verwendung von D und A/B in ein und derselben Routine sollte man immer besonders darauf achten, ob die gleichzeitige Verwendung der 8-Bit und 16-Bit Schreibweise der Akkus irgendwo störend interferiert. Im oben genannten Beispiel machen wir uns jedoch die Vorteile der alternativen 16-Bit Adressierung der Akkus zunutze. Die sparsame Variante sieht wie folgt aus:

```
1: 2000            ABC     EQU     $2000
2: 2010            XYZ     EQU     $2010
3: 0000
4: 0000 CC225B             LDD  #$225B
5: 0003 B72010             STAA XYZ
6: 0006 F72000             STAB ABC
```

Auf jeden Fall sollte man bei umfangreichen Initialisierungen prüfen, ob nicht eine Initialisierungstabelle verwendet werden kann. Eine solche Tabelle ist nicht nur übersichtlicher und einfacher zu warten, sie spart in vielen Fällen auch noch etliche Bytes Speicherplatz. Wie umfangreich die Einsparungen ausfallen, kann nicht pauschal gesagt werden, hier helfen nur eigene Versuche weiter.

1.5. HC11-spezifische Sonderbefehle

Durchaus kein Geheimnis, aber von Assemblerprogrammierern stets (und zu Unrecht) argwöhnisch beäugt sind die HC11-spezifischen Befehle für Bitmanipulationen und -tests. Gerade durch konsequenten Einsatz dieser Befehle lassen sich aber ganz deutliche Sparmaßnahmen im Code durchsetzen. Betrachten wir zuerst die Möglichkeiten, bestimmte Bits zu setzen bzw. zu löschen. Formulierungen wie die folgende sollten beim HC11 eigentlich nicht mehr auftauchen:

```
1: 1000            PORTA   EQU $1000
2: 0000
3: 0000 B61000             LDAA PORTA
4: 0003 8A20               ORAA #%00100000
5: 0005 B71000             STAA PORTA
6: 0008                    ; ...
7: 0008 9680               LDAA <$80
8: 000A 84D7               ANDA #%11010111
9: 000C 9780               STAA <$80
```

Stattdessen empfiehlt sich die Verwendung der Befehle BSET bzw. BCLR. Beide Befehle sind ausschließlich in den Varianten Direct und Indexed Addressing verfügbar. Der an den Befehl angehängte Mask-Value kennzeichnet das bzw. die Bit(s), welche(s) gesetzt/gelöscht werden soll. Es ist also durchaus möglich, nicht nur ein, sondern gleichzeitig mehrere Bits im Operanden zu beeinflussen. Der HC11 führt dazu eine Leseoperation auf den Operanden aus, verknüpft das gewonnene Byte mit der Maske (ODER bei BSET, AND mit dem Einerkomplement der Maske bei BCLR) und schreibt das Resultat zurück.

Aber Vorsicht! Denken Sie immer an die zuerst durchgeführte Leseoperation. Wenn Sie die Befehle BSET/BCLR auf Write-Only Register anwenden (Hardwareregister, z.B. ein

Ausgabelatch zur Ansteuerung von Relais) sind die Resultate nicht vorhersehbar (und in den seltensten Fällen erwünscht). Nach dieser langen Vorrede das modifizierte Beispielprogramm (es wird unterstellt, das X-Register sei bereits mit REGBAS $1000 geladen):

```
 1: 0000            xPORTA  EQU $00
 2: 0000
 3: 0000 1C0020             BSET xPORTA,x %00100000
 4: 0003                    ; ...
 5: 0003 1580D7             BCLR $80 %11010111
```

Eine Anmerkung zur Syntax: Die Maske ist zwar ein Immediate-Wert (wird vom Assembler 1:1 im Code eingefügt), das führende Doppelkreuz entfällt hier jedoch, da im Gegensatz zu anderen Befehlen an dieser Stelle gar keine andere Adressierungsvariante in Frage kommt. Ebenso wird das Kleiner-Als Zeichen als Kennzeichen für Direct Addressing hier dem Operanden nicht vorangestellt, da Extended Addressing an dieser Stelle nicht in Frage kommt. In diesen Details gibt es jedoch von Assembler zu Assembler unterschiedliche Interpretationen und Regeln.

Eine weitere Möglichkeit, einige Bytes einzusparen, bieten die bedingten Sprungbefehle BRSET (Branch if bit(s) set) und BRCLR (Branch if bit(s) clear). Syntaktisch sind sie den oben genannten Bitmanipulationsbefehlen BSET und BCLR vergleichbar, es kommt jedoch noch ein weiterer Operand hinzu, der das Sprungziel bei positivem Entscheid festlegt. Dieses Sprungziel muß im Bereich -128...127 ausgehend von der Adresse der nachfolgenden Anweisung liegen, da es sich um einen Relativsprung handelt, der im Programmcode mit einem einzigen vorzeichenbehafteten Offsetbyte definiert wird. Zuerst wieder ein Beispiel für klassische, aber unelegante Programmierung:

```
 1: 102E            SCSR    EQU $102E
 2: 102F            SCDR    EQU $102F
 3: 0000
 4: 0000 B6102E     getSCI  LDAA SCSR
 5: 0003 8420               ANDA #$20
 6: 0005 27F9               BEQ  getSCI
 7: 0007 B6102F             LDAA SCDR
 8: 000A 39                 RTS
 9: 000B
10: 000B 36         putSCI  PSHA
11: 000C B6102E     _wtTDRE LDAA SCSR
```

```
12: 000F 8480                ANDA #$80
13: 0011 27F9                BEQ  _wtTDRE
14: 0013 32                  PULA
15: 0014 B7102F              STAA SCDR
16: 0017 39                  RTS
```

Substituiert man die Test- und Sprungbefehle des obigen Beispiels mit BRSET/BRCLR-Anweisungen, erhöht sich die Lesbarkeit und gleichzeitig wird weniger Speicherplatz belegt:

```
 1: 1000            REGBAS  EQU $1000
 2: 002E            xSCSR   EQU $2E
 3: 002F            xSCDR   EQU $2F
 4: 0000
 5: 0000                    ; ...
 6: 0000 CE1000             LDX   #REGBAS
 7: 0003                    ; ...
 8: 0003
 9: 0003 1F2E20FC   getSCI  BRCLR xSCSR,x $20 getSCI
10: 0007 A62F               LDAA  xSCDR,x
11: 0009 39                 RTS
12: 000A
13: 000A 1F2E80FC   putSCI  BRCLR xSCSR,x $80 putSCI
14: 000E A72F               STAA  xSCDR,x
15: 0010 39                 RTS
```

Eine "nette" Anweisung ist auch die TST Instruction, die in vielen Programmen unfairerweise eher ein Schattendasein fristet. Dabei ist es gar nicht so selten, daß man wissen möchte, ob ein Wert Null ist, oder was mit dem Bit 7 gerade so los ist. Also, wenn Sie wiedermal das verflixte siebte Bit quält erinnern Sie sich vielleicht an den TST Befehl. Übertreiben darf man jedoch auch den Einsatz dieser Spezialbefehle nicht, man sollte von Fall zu Fall neu abwägen, ob sich die eine oder andere Variante günstiger gestaltet.

1.6. Sprünge und Unterprogramme

Der HC11 kennt absolute (JMP) und relative (BRA) Sprünge (An dieser Stelle sei die Bemerkung gestattet, daß es stets um die Frage geht, wie weit der HC11 springt, nicht wie hoch). Relative Sprünge bieten den Vorteil, daß sie ein Codebyte weniger als das absolute Pendant erzeugen, wenn man die Verwendung der Adressierungsart Extended unterstellt. Da ihr Wirkungsbereich aber eingeschränkt ist, kommt man nicht immer einzig und allein mit relativen Sprüngen zurecht. Scheint die Lage dementsprechend aussichtslos, lohnt es sich

zu prüfen, ob man das Ziel über Umwege dennoch erreichen kann. Findet man im Programm eine weitere Verzweigung zum gewünschten Sprungziel, kann man diese ggf. als Zwischenstopp benutzen. Liegt die Verzweigung in der Reichweite eines BRAnch Befehls, so kann man zuerst zu der Verzweigung springen, die dann wiederum zum eigentlichen Sprungziel führt. Das Springen in Codebereiche, die mit der aktuellen Funktion ursprünglich nichts zu tun haben, ist zwar nicht die feine englische Art, aber schließlich ringen wir ja verzweifelt um jedes Byte.

Ähnliche Sparmaßnahmen kann man bei den Unterprogramm-Verzweigungen versuchen. Ein Unterprogrammaufruf der Form JSR xxxx erzeugt drei Byte Code. Liegt das Unterprogramm "in Reichweite", also im Bereich -128 bis +127, ausgehend von der Adresse des unmittelbar folgenden Befehlscodes, dann bedient man sich stattdessen vorteilhafter Weise der verkürzten, relativ adressierten BSR Variante, welche mit zwei Codebytes auskommt. Nun mag es vorkommen, daß ein Unterprogramm von mehreren Stellen im Programm aufgerufen werden soll, aber zumindest eine Stelle zuweit von dem Unterprogramm entfernt liegt. Durch geschickte Anordnung der einzelnen Routinen eines Programms ist es jedoch oftmals möglich, einige JSR Befehle doch noch durch BSR zu ersetzen. Probieren geht hier im konkreten Fall wieder über studieren.

Kommt es dazu, daß ein Unterprogramm sehr häufig verwendet werden soll, so lohnt es sich u.U., das derart beliebte Unterprogramm als Interruptroutine zu formulieren. Einerseits spart man sich bei dieser Methode einige Bytes, die sonst für das Retten und Restaurieren von Registern erforderlich gewesen wären. Diese Tätigkeiten werden vom Prozessor bei Eintritt in einen Interrupt selbsttätig ausgeführt. Ebenso findet man nach Rückkehr in das Hauptprogramm automatisch alle Register wieder in unversehrter Form vor. Der eigentliche Trick besteht aber darin, den SWI (Software Interrupt) Befehl des HC11 zu verwenden, der nur ein einziges Codebyte benötigt. Jeder Unterprogrammaufruf, der mit SWI ausgeführt wird, spart demzufolge ein (BSR) oder zwei (JSR) Byte ein. Erkaufen muß man sich den Vorteil mit einer Initialisierungssequenz für den SWI-Interrupt. Schließlich ist es notwendig, einen entsprechenden Interruptvektor zu installieren

und (wenn nicht ohnehin geschehen) das globale Interrupt-Maskierungsbit freizugeben. Das folgende Listing zeigt die notwendigen Schritte im Special Bootstrap Mode:

```
1: 00F4            swivec  equ $00f4
2: 0000
3: 0000 867E       prepare ldaa #$7e
4: 0002 97F4               staa <swivec
5: 0004 CC000A             ldd  #sub1
6: 0007 DDF5               std  <swivec+1
7: 0009 0E                 cli
8: 000A
9: 000A 3B         sub1    rti
```

Fazit: Nach spätestens 10 Unterprogrammaufrufen hat sich die Mühe gelohnt, evtl. aber auch schon vorher. Zum Schluß dieses Abschnitts sei darauf hingewiesen, dem Unterprogramm einige Beachtung zu schenken: Vergessen Sie nicht, die Routine mit RTI statt RTS abzuschließen! Verschwiegen werden soll auch nicht, daß die Methode via Interrupt einige Zeit in Anspruch nimmt. Während BSR/RTS zusammen 11 Takte benötigen, nehmen SWI/RTI 26 Takte in Anspruch. Das ist die Kehrseite der Medaille.

1.7. Special Bootstrap Mode

Der HC11 läßt sich in vier verschiedenen Modi betreiben. Werden in einer Applikation keine externe Speicher angeschlossen, ist die bevorzugte Betriebsart der Special Bootstrap Mode. Er unterscheidet sich vom Expanded Mode unter anderem dadurch, daß automatisch eine Firmware Routine, welche sich im Adreßbereich $BF40..$BFFF in einem entsprechenden ROM-Bereich befindet, ausgeführt wird. Diese Routine wird i.Allg. dazu verwendet, um entweder ein Programm in den internen RAM zu laden, oder aber um ein im internen EEPROM vorhandenes Programm zu starten. Da dieser Ablauf reproduzierbar in jedem HC11 stattfindet, kann man auch programmtechnisch darauf aufbauen. Ziel ist natürlich, wieder ein paar Byte Programm einzusparen. Zum besseren Verständnis folgt an dieser Stelle ein gekürzter Auszug aus der Bootstrap-Loader Firmware eines 68HC11A8:

```
*************************************************
* BOOT
* BOOTLOADER FIRMWARE FOR 68HC11A8 W/O SECURITY
* EQUATES FOR USE WITH INDEX OFFSET = $1000
*
PORTD   EQU     $08
DDRD    EQU     $09
SPCR    EQU     $28     (FOR DWOM BIT)
BAUD    EQU     $2B
SCCR1   EQU     $2C
SCCR2   EQU     $2D
SCSR    EQU     $2E
SCDAT   EQU     $2F
PPROG   EQU     $3B
TEST1   EQU     $3E
CONFIG  EQU     $3F
*
* MORE EQUATES
*
EEPSTR  EQU     $B600           START OF EEPROM
EEPEND  EQU     $B7FF           END OF EEPROM
*************************************************
        ORG     $BF40
*
BEGIN   EQU     *
* INIT STACK
        LDS     #$00FF
* INIT X REG FOR INDEXED ACCESS TO REGISTERS
        LDX     #$1000
* PUT PORT D IN WIRE OR MODE
        BSET    SPCR,X $20
* INIT SCI AND RESTART BAUD DIVIDER CHAIN
        LDAA    #$A2            DIV BY 16
        STAA    BAUD,X
* RECEIVER & TRANSMITTER ENABLED
        LDAA    #$0C
        STAA    SCCR2,X
        ...
        ...
*************************************************
*
        ORG     $BFD4           NEEDED IF BOOTROM < MAX
* MASK I. D. BYTE
        FDB     $0000
*
*************************************************
* VECTORS
*
        FDB     $100-60         SCI
        FDB     $100-57         SPI
        ...
        FDB     $100-3          CLOCK MONITOR
        FDB     #BEGIN          RESET
*************************************************
```

Ein näherer Blick in das Listing ergibt folgende Erkenntnisse: Der Stackpointer wird auf das Ende des internen RAM gesetzt, das Indexregister X zeigt auf die Basisadresse des Registerbereichs, Port D ist auf Open-Drain Mode geschaltet und das SCI wird auf 7812 Baud initialisiert. Anschließend wird der Sender und der Empfänger des SCI freigegeben. Das sind alles ziemlich vernünftige Defaults, um die man sich im eigenen Anwenderprogramm nicht nochmals kümmern muß. Dadurch kann man ohne große Mühe einige 10 Byte einsparen - und Uneingeweihte mit "gekappten" Programmen auch ein wenig verblüffen.

1.8. Auslagern von Code in serielle Speicher

Wenn es eng wird im Gehäuse, ist man bestrebt, Programmteile, die partout nicht mehr in den internen EEPROM des HC11 passen wollen, in externe Speicher zu verlagern. Berücksichtigt man die Notwendigkeit externer EPROMs oder RAMs bereits beim Entwurf, kann man in aller Ruhe an die 64 KB Grenze heran programmieren. Lösungen mit externen (parallelen) Speicherbausteinen belegen aber zwei Ports des HC11 und kosten mehr Geld, als vielleicht unbedingt notwendig. Die zwei verlorenen Ports (Port B und Port C) kann man zwar mit einer 68HC24 Port Replacement Unit oder einigen Latches zurückgewinnen, das verursacht aber erneut Zusatzkosten und es wird außerdem eng auf der geplanten Platine.

Ein Ausweg stellen serielle Speicher dar. Sie benötigen nur zwei Bit zur Kommunikation mit dem Mikrocontroller. Zwei Bit sind in aller Regel einfacher "aufzutreiben" als zwei ganze Ports für die übliche parallele Variante. Zudem sind serielle PROMs, wie der im folgenden Applikationsbeispiel verwendete XICOR Baustein, in wesentlich kompakteren Gehäusen untergebracht. Üblich sind nicht nur 8-polige DIP Gehäuse, auch SMD Bausteine sind in allerlei Varianten erhältlich. Serielle Speicherbausteine stellen somit eine Ideallösung bezüglich Ressourcenverbrauch und Platzbedarf dar - wenn da nicht doch noch ein Problem wäre. Dieses Problem besteht darin, daß der in einem seriellen PROM enthaltene Programmcode vom HC11 nicht direkt abgearbeitet werden kann. Der Controller holt sich Instructions und Daten nur parallel

von internen oder externen Speicherbereichen. Die serielle Variante wäre normalerweise auch nicht in den gewünschten Geschwindigkeiten durchführbar. Immerhin hat der HC11 einen Bustakt von 2 MHz bei einer Datenbusbreite von 8 Bit, somit ergäbe sich ein Bittakt von mindestens 16 MHz. Dieser theoretische Wert reicht aber noch nicht aus - ohne weitere genaue Berechnungen anzustellen sieht man jedoch, daß der Zugriff auf serielle Speicherbereiche normalerweise Probleme aufwirft und daher nicht als Standardzugriffsvariante in den üblichen Mikrocontrollern implementiert ist.

Zurück zu den Vorteilen serieller Speicher und unserem Vorhaben, Code oder Daten auszulagern. Um auf Code oder konstante Daten in diesen Speichern zugreifen zu können, benötigt man ein kleines Overlay-Konzept. Geht man davon aus, daß der interne RAM meistens nicht zu hundert Prozent ausgelastet ist, findet man hier auch Platz für die Ablage von Programmroutinen oder (zeitweise benötigten) Datenbereichen. RAM eignet sich von Haus aus nicht gerade zum Ablegen von dauerhaften Informationen, da er bei Strommangel stark an Gedächtnisverlust leidet. Lädt man den RAM aber bei Bedarf mit Daten aus einem seriellen "Byte-Lager", so hat man die Daten bzw. den Programmcode anschließend im RAM im zugreifbaren/abarbeitbaren Zustand. Je nach Programmsituation kann man den gleichen Pufferbereich im RAM auch mit unterschiedlichen Daten aus dem seriellen PROM belegen. Der dazu benötigte "Overlay-Manager" kann spartanisch ausfallen, belastet also den knappen Speicherhaushalt nicht noch zusätzlich.

Nachdem die zugrundeliegende Idee soweit deutlich geworden sein dürfte, folgt ein Schaltungsvorschlag zur Ankopplung eines seriellen PROMs an den HC11. Abb. 1-1 zeigt den Schaltplan, wobei alle für das Verständnis unwesentlichen Beschaltungen des HC11 (IC1) weggelassen wurden.

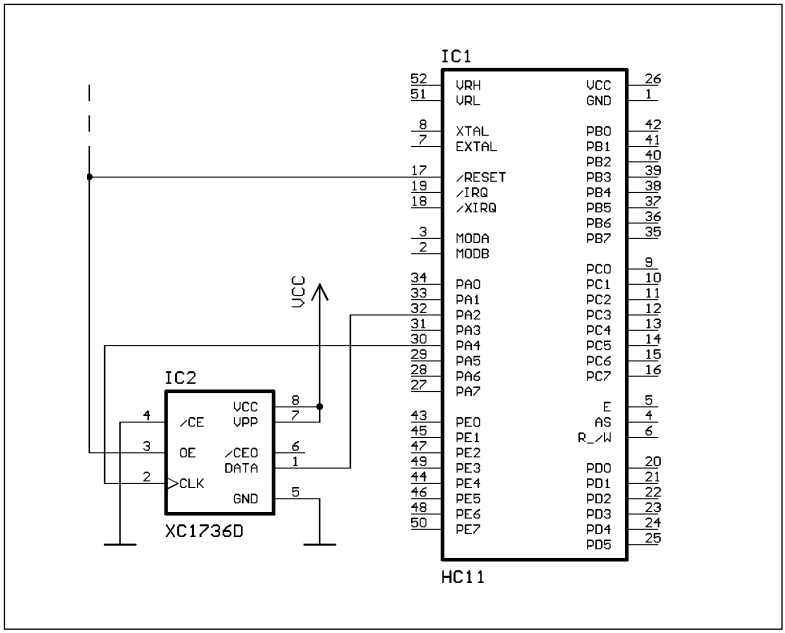

Abb. 1-1: Anschluß eines seriellen PROMs an den HC11

Für die Ansteuerung des sPROM wurden mehr oder weniger willkürlich die Portleitungen PA2 und PA4 ausgewählt. Denkbar wären noch eine ganze Menge anderer Portkombinationen, inklusive der Ankopplung über das SPI-Interface. Der Speicherbaustein (IC2) ist ein Typ mit einer Kapazität von 36 KBit von XICOR. Es handelt sich um einen OTP Baustein, d.h. er ist nur einmal programmierbar. Alternativ lassen sich auf ähnliche Weise selbstverständlich auch serielle EEPROM Bausteine einsetzen, welche elektrisch löschbar und dadurch mehrfach programmierbar sind.

Einige Erläuterungen zu den Funktionen der einzelnen Anschlüsse von IC2: Über VCC und GND wird der Chip mit der Betriebsspannung (5 Volt) versorgt. Wichtig ist die Verbindung der Programmierspannung VPP zu VCC. VPP muß im Betrieb stets mit der Betriebsspannung verbunden sein, sonst kann es zu schwer auffindbaren Fehlfunktionen kommen.

Der Chip Enable Pin /CE liegt permanent auf L-Pegel. Um die Stromaufnahme zu senken, wenn der Baustein zeitweise

nicht benötigt wird, kann man diesen Pin auf H-Pegel legen. Dazu wäre dann eine weitere Steuerleitung vom HC11 bereitzustellen. In der gezeigten Beispielapplikation wird von dieser Möglichkeit kein Gebrauch gemacht.

Der Anschluß OE (Output Enable) gibt nicht nur den Datenausgang frei, durch L-Pegel an diesem Anschluß wird auch der interne Adreßzähler des sPROM zurückgesetzt. Will man innerhalb eines Programms mehrmals den gleichen Speicherbereich aus dem sPROM auslesen, sollte man den OE Pin nicht wie gezeigt mit dem /RESET Signal verbinden, sondern diesen Pin über eine Portleitung des HC11 steuern. Am Rande sei bemerkt, daß sich die Polarität dieses Pins programmieren läßt. Es kann also durchaus sein, daß derselbe Anschluß in einer anderen Applikation mit vorangestelltem Negations-Slash (/OE) bezeichnet wird. Dies ist also kein Fehler, es ist lediglich eine Frage der Konfiguration. Festgelegt wird die Polarität beim Programmieren des Bausteins mit einem geeigneten Programmiergerät (für die gängigen Universalprogrammierer, wie z.B. den ALL-07, kein Problem).

CLK ist der Takteingang des Speicherchips. Mit jeder L-H-Flanke wird der interne Adreßzähler um Eins weitergeschaltet und das folgende Bit erscheint am Datenausgang DATA.

/CEO ist ein Ausgang, der zur Ansteuerung weiterer Speicherbausteine vom gleichen Typ dient. Dadurch ist es möglich, mehrere Bausteine zu kaskadieren, um die Gesamtkapazität zu erhöhen. Da es neben dem XC1736 mit 36 KBit (bzw. 4,5 KByte) auch noch Typen wie den XC1765 mit 65 KBit, oder gar den XC17128 mit 128 KBit gibt, wird man diese Kaskadierungsoption nur in seltenen Fällen einsetzen müssen. Daher sei an dieser Stelle auch nicht weiter auf diese Möglichkeit eingegangen.

Soweit zur Hardware, wie man sieht, gestaltet sich der Anschluß eines seriellen Speichers an den Mikrocontroller recht einfach. Nun ein Blick auf die nicht minder wichtige Software. Das folgende Listing zeigt eine kurze Routine, die (aufbauend auf die Hardwaregegebenheiten in Abb. 1-1) einen Speicherblock vom sPROM in den RAM transferiert. Die Startadresse des zu belegenden RAM-Bereichs erhält die Routine über das X-Register mitgeteilt, die Länge des Blocks steht in

Accu B (1..256 Byte, wobei der Wert $00 im Accu B den Fall 256 Byte repräsentiert):

```
;*************************************************************;
; File: K1SPROM.A                                             ;
; Date: 26.05.95                                              ;
; Func: Move sPROM to RAM                                     ;
; Hard: Data from sPROM PA2, Clock to sPROM PA4               ;
;*************************************************************;
REGBAS          equ     $1000
yPORTA          equ     $00
ROMCLK          equ     $10
ROMDAT          equ     $04
; Func: Move Data from sPROM to RAM area
; Args: X = Pointer to RAM buffer
;       B = Size of RAM Buffer
; Retn: -
;
moveSPROM       psha
                pshb
                pshx
                pshy
                ldy     #REGBAS
_nextByte       ldaa    #8                      ; 8 Bits in a Byte
_nextBit        lsr     0,x                     ; Bit 7 := 0
                brclr   yPORTA,y ROMDAT _clockIt
                bset    0,x $80                 ; Bit 7 := 1
_clockIt        bclr    yPORTA,y ROMCLK
                bset    yPORTA,y ROMCLK         ; CLK: L->H
                deca
                bne     _nextBit
                inx
                decb
                bne     _nextByte
                puly
                pulx
                pulb
                pula
                rts
;-- fine -------------------------------------------------------;
```

1.9. Zusätzliche RAM-Zellen in Peripheriebausteinen

Auf der Suche nach zusätzlichen RAM Bytes muß man u.U. auch unkonventionelle Wege gehen. Stilistisch gesehen zählt es zwar nicht zu den Tugenden eines Programmierers, wenn er seine Programme mit exotischen, schwer nachvollziehbaren Konstrukten spickt. Andererseits will auch nicht jeder Softwareautor, daß seine Programme von Jedermann durchschaut (... und kopiert...) werden. Insofern ergeben sich

auch Möglichkeiten, einen gewissen Kopierschutz in eigene Applikationen einzubauen. Solch ein Kopierschutz könnte z.B. darauf beruhen, daß man bestimmte Variablen in RAM Zellen von Peripheriebausteinen ablegt. Verwendet eine ähnliche Schaltung nicht genau denselben Peripheriebaustein, wird die Software darauf nicht lauffähig sein.

Bei den hier angestellten Überlegungen steht aber das Auffinden von zusätzlichen Speicherbereichen an erster Stelle. Diesbezüglich fündig wird man an sehr vielen Stellen. Gern eingesetzt werden z.B. RTC's, das sind Uhrenchips mit eigenem Zeitnormal und einem Satz Register zur Ablage der Zeitinformation. Neben der obligatorischen Grundausstattung Zählregistern für Sekunden, Minuten, Stunden, Tag, Monat, Jahr und Wochentag ist es bei den diversen Herstellern auch üblich, in solchen Chips zusätzliche Notiz-Register einzubauen. Legt man Informationen in diesen Notizregistern, oder vielleicht einfach in den Datum Registern (so man von der RTC nur die Zeit wissen will) ab, ergibt sich zudem der Vorteil, daß die Zellen i.d.R. batteriegepuffert, also nichtflüchtig, sind.

Auch LC-Displays verwöhnen den Anwender mit zusätzlichem Speicher. Die gängigen alphanumerischen Displays mit Eigenintelligenz (gefertigt von Philips, Sharp, Optrex usw. usf.) haben einen eigenen Bildschirmspeicher und einen RAM-Bereich zur Ablage einiger benutzerdefinierter Zeichen (sog. CG-RAM). Kommt man mit dem Standardzeichensatz aus, der sich im Character Generator ROM des Displays befindet, kann man den CG-RAM anderweitig verwenden. Durch den üblicherweise auf den LCD's eingebauten Controllerchip HD44780 kann man sowohl lesend als auch schreibend auf den CG-RAM zugreifen. Durch einigen Kommunikationsoverhead sind die Zugriffe zwar nicht sehr schnell, die Zugriffszeit liegt in der Größenordnung 50 bis 100 µs. Dafür ist der gewonnene Platz mit 64 Byte mehr als üppig. Hinzu kommen, je nach verwendetem Modul, noch ungenutzte Bereiche im DD-RAM (Display Data RAM, Bildschirmspeicher).

Die beiden Beispiele sollten aufzeigen, daß es sich lohnt, ohnehin eingesetzte Peripheriebausteine genauer unter die

Lupe zu nehmen. Oft ist man positiv überrascht, welche Möglichkeiten noch ungenutzt in ihnen schlummern.

1.10. Steuerregister als Notizzettel

Aufmerksam geworden auf diese Möglichkeit ist der Autor beim Studium des Bootloaders des 68HC11E9. Dort ist im EQUates Abschnitt das TOC1 Register mit aufgeführt, obwohl dieses für den Bootstrap Vorgang keine Rolle spielt. Kommentiert ist die Anweisung mit "Extra Storage, Poor Style" (siehe [1], Seite B-17). Mit diesem Kommentar sind beide Seiten der Idee hinreichend charakterisiert: Zusätzlicher Speicherplatz, aber armseliger Stil. Zur Stilistik sei auf die Ausführungen im vorangegangenen Abschnitt verwiesen. Uns interessiert wiederum vorrangig, wie wir noch ein paar zusätzliche Byte zusammenklauben können. Das TOC1 Register bringt alle Voraussetzungen mit, die es für solche Zwecke aufweisen muß: Es ist möglich, darauf zu schreiben und davon zu lesen, es verändert seinen Inhalt nicht "willkürlich" und Zugriffe auf das Register sind (normalerweise) nicht mit schädlichen Nebenwirkungen verbunden.

Neben dem TOC1 Doppelregister (16 Bit) kann man selbstverständlich auch die anderen Timer Output Compare Register (TOC2 bis TOC5) in analoger Art und Weise verwenden. Nicht geeignet sind hingegen die Timer Input Capture Register TIC1 bis TIC3, da es sich hierbei um Read-Only Register handelt. Besser sieht es mit dem 8-Bit Register PACNT aus, dieses ist jederzeit schreib- und lesbar. Allerdings muß man immer die Einschränkungen in Betracht ziehen, die sich bei der Zweckentfremdung von Steuerregistern ergeben. Daher sollte man vorher wirklich *sehr* genau wissen, wie das umgenutzte Register arbeitet und welche Randbedingungen stimmen müssen. Allen Neu-Usern daher der Tip: Vorerst Hände weg von akrobatischen Programmiertricks.

1.11. Entwicklungsumgebung

Alle vorgestellten Programmbeispiele wurden mit der HC11 Entwicklungsumgebung IDE11 übersetzt [5]. Die einzige Ausnahme davon ist das Teile des HC11 Bootloaders zeigende Listing, welches aus dem Motorola Reference Manual [1] übernommen wurde. Die Sharewareversion der IDE11 Entwicklungsumgebung steht allen Interessenten kostenlos zum Download bereit. Zu finden ist die Software (IDE11.EXE) im Area "Freeware/Shareware" der folgenden Mailbox:

EMUF EPAC BBS Detmold
(05232) 85112

Die Übertragungsparameter sind, wie üblich, 8N1 mit maximal 14400 bps. MNP5 und V42.bis werden unterstützt. Melden Sie sich mit Ihrem realen Vor- und Zunamen an, wechseln Sie in das o.g. Area und wählen Sie den Menüpunkt Download. Als Übertragungsprotokoll empfiehlt sich Z-Modem. Über aktuelle Möglichkeiten zum Download über das Internet wenden Sie sich bitte an den Autor.

1.12. Literatur

[1] HC11 Reference Manual, Rev. 3; Motorola Inc., 1991; Part No. M68HC11RM/AD

[2] MC68HC11A8 Technical Data, Rev 6; Motorola Inc., 1994; Part No. MC68HC11A8/D.

[3] MC68HC811E2 Technical Data; Motorola Inc., 1991; Part No. MC68HC811E2/D.

[4] Sturm, Matthias: Elektronik-Aufgaben Band III: Mikrorechentechnik; Leipzig; Fachbuchverlag Leipzig GmbH, 1994.

[5] Thamm, Oliver: IDE11 Integrierte Entwicklungsumgebung für 68HC11, V2.2, Benutzerhandbuch; Leipzig; MCT Lange & Thamm Mikrocomputertechnik GbR, 1995